Summer 2010

A Design of a Generic Profile-Based Queue System

Ali Husain
Regis University

Regis University
College for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

A DESIGN OF A GENERIC PROFILE-BASED QUEUE SYSTEM

A PROJECT

SUBMITTED ON THE 6TH OF MAY,2010

TO THE DEPARTMENT OF INFORMATION TECHNOLOGY

OF THE SCHOOL OF COMPUTER & INFORMATION SCIENCES

OF REGIS UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF MASTER OF

SCIENCE IN DATABASE TECHNOLOGIES

BY

Ali Husain

APPROVALS

Charles Thies, Project Advisor

Shari Plantz-Masters

Richard L. Blumenthal

Abstract

Website and server hosting accounts impose resource limits which restrict the processing power available to applications. One technique to bypass these restrictions is to split up large jobs into smaller tasks that can then be queued and processed task by task. This is a fairly common need. However, different application jobs can differ widely in nature and in their requirements. Thus, a queue system built for one job type may not be entirely suitable for another. This situation could result in the having to implement separate, additional queue systems for different needs. This research proposes a generic queue core design that can accommodate a large variety of job types by providing a basic set of features which can be easily extended to add specificity. The design includes a detailed discussion on queue implementation, scheduling, directory structure and business tier logic. Furthermore, it features highly configurable, time-sensitive performance management that can be customized for any job type. This is provided as the ability to indicate desired performance profiles for any given slot of time during the week. Actual performance data based on the usage of a prototype is also included to demonstrate the significant advantage of using the queue system.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Background Information

Web applications can be hosted on premises or outsourced to a hosting company. The benefit of outsourcing is that many non-business tasks are handled efficiently and easily such as backups, server maintenance, technical support, etc. The alternative is to maintain a potentially much more expensive IT team and infrastructure to support the web application. Several hosting options exist with many variations. In general, however, small to medium organizations can either purchase a website or server hosting account. A website hosting account provides space on a shared server while a server hosting account provides a full interface to a virtual or dedicated server. However, the major drawback of hosting accounts is that server resources are restricted such as CPU cycles, bandwidth, emails per hour, memory and other resources. Resources can only be increased by paying a higher price. If a web application is not performing satisfactorily and refactoring the code is not an option, it could mean that the hosting package must be upgraded to a higher-priced package. If upgrading to a less restrictive account is not an option because of cost, either trade-offs must be made or other solutions found.

One solution is enhancing system design to reduce resource consumption. Careful design can result in an application that is scalable yet intelligent and conservative in its consumption of resources. Design decisions at all application layers can have a significant impact on resource usage. (Controlling Resource Consumption and Improving Performance, 2007). A simple example is that many operations that used to require page reloads are now handled using AJAX. It does not require the page to be reloaded and usually performs minimal transfers of text between the client and server which speeds up

applications. This greatly reduces bandwidth consumption. Thus, AJAX is actually saving subscribers hosting costs. Therefore, with minimal resource usage kept as a design goal, hosting costs can be greatly reduced by making proper design decisions.

A specific related scenario is that of the university department in which this research took place. It had a limited IT budget yet had to provide services for a large number of students. Suppose the department mentioned must create thousands of individualized score reports and email them to students. Emailing them at the moment of report creation is not be feasible especially if the mail server is located on a different server and network which means a significant delay is incurred with every connection to this server. This poses a clear resource problem. How can this department provide web services to thousands of students while minimizing hosting costs? Furthermore, if all students must receive weekly emails, how can this be done in a timely manner if only a limited number of emails are allowed to be sent per hour? The problem is equally applicable to any small organization with similar budget constraints and hosting needs such as a community organization, a private school, etc.

The logical solution is to implement a queue system to perform these tasks gradually to remain within resource limits. Individualized student reports can be stored in a queue that can be processed at regular intervals. In this manner, large amounts of limited resources can be consumed over time. Software queues, especially for emailing, are very common. Queues can be useful for many other types of tasks as well. In this particular department, other such tasks included processing thousands of enrollments and importing and validating thousands of scores from files. However, it quickly became apparent to this department that implementing a queue for every new type of task was a

waste of labor resources. Instead, the department envisioned a single generic queue infrastructure that could be extended to accommodate any type of task.

Another issue regarding queues is throughput. If total throughput is the only performance goal considered, resources could be exhausted and users could experience slow system responses because the queue is consuming too much. Ideally, the queue should be able to adjust its resource usage based on pre-configured resource consumption parameters based on day of week and time. For example, during non-working hours in the university scenario, usage of the system drops dramatically. After midnight, for example, the system is usually not used at all. Thus, it makes sense to boost queue processing during such periods and to reduce it during peak usage times. Another way to state this is that different periods of time based on day of week and time of day should possess different processing *profiles*.

Thus this department endeavored to research and implement the requirements and design of a highly flexible queue system that was generic and profile-based. It was to be generic in that it could be extended to accommodate any task type. Each task type had its own set of processing profiles to maximize resource usage without exceeded limits placed by the hosting package.

The inspiration for this research came from a previous queue-based email system developed for the university department. This system, however, was rudimentary and not generic. Rather, it was built specifically for emailing with hard-coded processing parameters. This research effort took this software and further developed it to meet the specifications discussed herein.

1.2 Statement of the Problem

With the pervasiveness of the Internet, applications are increasingly becoming web-based. However, hosting accounts place impose several resource restrictions such as execution time and memory allocated. Therefore, certain tasks that are easily processed in less restricted desktop or network applications must be processed gradually in their web-based versions in order to stay within resource limits. Furthermore, complex applications can have several types of these long-running tasks or jobs. How can a *single*, generic system be developed to store and gradually process *any* type of job? Also, how can the rate of gradual processing be optimally controlled such that the system utilizes resources more aggressively during low usage periods in order to finish processing jobs faster?

1.3 Statement of Goals

This research effort seeks to develop a modular, generic and extensible queue design that processes multiple types of jobs and tasks over time based on configurable resource consumption parameters. These parameters are defined in time-based profiles. The ultimate goal is to be able to schedule tasks for gradual processing that stays within allotted resource limits and user deadlines. The intended fruit of such research is that application hosting costs can be kept lower by spreading out resource usage over time. Furthermore, full design details are included, explained and discussed such that any reader can apply the same design in a different context. It is the claim of this thesis that basing queue processing on these profiles can achieve acceptable throughput, decrease hosting costs, minimize disruption to users and maximize general efficiency.

Chapter 2 - Review of Literature and Research

A comprehensive effort was made to study existing literature related to queues, scheduling, web application architecture and related disciplines. This was done in order to gain a broad understanding of queue implementation issues and to avoid repetition of previous research. Furthermore, useful concepts found in that literature were later incorporated into this research. Special consideration was given to investigating issues not fully addressed by the current literature pool.

Literature was analyzed from a wide variety of fields. This is a natural consequence of the fact that queues are a general concept applicable to many areas of research. Thus, any research pertaining to queues can potentially benefit from a completely different field of research that includes the concepts, theory and principles related to queues. Some prominent areas of queue research are the fields of circuit and processor engineering, network packet handling, web server request handling, manufacturing, queue theory and scheduling. Many of these areas were investigated and are categorized and described below.

2.1 Application Performance

An application's performance is depends on many factors. However, the specifications of the hosting server play a large role in determining the user's perceived performance of the application. In general, a server with faster processors and more memory can respond faster than a slower, less powerful server.

However, this is true only up to a certain point (Slothouber, 1996). In fact, carelessly using more hardware can actually decrease the overall response time (Slothouber, 1996). Furthermore, it is not always a feasible solution to invest in more hardware since server

specifications are limited by financial constraints. Thereafter, the software aspect of the architecture must be made more efficient.

No matter how powerful the server or server cluster, it is an accepted fact that today's web servers and the applications that they serve perform poorly under a heavy load. A dramatic deterioration in server response-time and connection quality occurs as requests to the server exceed the allowed capacity (Abdelzaher & Bhatti, 1999).

Furthermore, as the load on the server increases so does the need for higher hardware specifications and additional servers. Thus, it is incumbent upon developers of busy websites to keep efficiency as a design goal in order to reduce the load on the server wherever possible. By doing so, costs associated with running and hosting the website can also be reduced. In addition, the perceived experience of the user is improved.

Several researchers have attempted to address the issue of application performance by offering various solutions and focusing on the optimization of different aspects of an application and its supporting environment to reduce the server load. Some of the work focused on optimizing the web server software on which the application runs. Others focused on server hardware and cluster configurations. Others focused on the application and more specifically, the different tiers of an application's architecture such as the data tier and business tier. Specific examples of these are mentioned below.

Among those who work was centered on server optimization were Abdelzaher and Bhatti (1999). The approach of these researchers was to reduce server load without causing a perceivable change to the user. They outlined a method to make the server intelligently modify content if an overload condition was detected and then to substitute the normal content with a "degraded, less resource intensive version of the requested

content" in order to reduce the size of the HTTP response (Abdelzaher & Bhatti, 1999, p. 1564).

The research of Abdelzaher & Bhatti was unique in that much previous research was concerned with different types of load balancing solutions. Instead, they limited their scope to what was occurring on an individual server. Abdelzaher and Bhatti (1999) proposed a number of ways to produce the lighter version of content such as compressing images, reduction of embedded objects per page, and reduction in the number of local links in the website that resulted in less browsing. The mechanism for exerting control over the content, which they described in detail, is an independent process separate from the web server which they call the *adaptation agent*. The adaptation agent monitors the server load and takes action to adapt content if it detects an overload condition. In their scenario the adaptation agent simply changed the properties of a directory link in the web document root to point to an alternative directory containing the lightweight content.

However, they note a major drawback to their method which is that content must be pre-processed and stored for later retrieval. It cannot be produced at the time of the request. Doing so, based on their previous research (Abdelzaher and Bhatti, 1999), only further increases the server load. Therefore, their methods are limited to static content and offer no advantage for dynamic content. (Abdelzaher and Bhatti, 1999).

Just as Abdelzaher and Bhatti achieved performance through using an entity external to both the application and the web server, other researchers also took this approach. These researchers investigated how modifications to the operating system kernel could be used to enhance performance or provide greater stability to an application. One group developed a loadable kernel module that could be used to control the rate of requests

being sent to the web server to avoid overload (Voigt, Tewari, Freimuth & Mehra, 2001). They demonstrated that this was more effective and scalable than application-level controls implemented within the web server software itself. This is a technique generally referred to as *admission control*. However, they note that success of this method is dependent upon highly accurate policies that determine which packets to drop. Furthermore, there is an underlying assumption "that the system administrator has complete understanding of server behavior under varying load conditions" (Voigt, Tewari, Freimuth & Mehra, 2001, p. 13). This requirement introduces difficulty into the implementation of this technique.

Harchol-Balter and Schroeder (2006) also criticized admission control since ultimately it can still result in denial of service to some users (Harchol-Balter & Schroeder, 2006). Therefore, other efforts were made to improve performance while not denying requests. An example is that of Harchol-Balter, Schroeder, Bansal and Agrawal (2003) in which they, like the researchers above, worked with the kernel. However, their solution was not filtering requests by controlling admission. Instead, it was modifying the way in which requests were scheduled. In other words, they investigated how to improve the order in which the HTTP requests in the queue are being processed.

In the intriguing research of Harchol-Balter, Schroeder, Bansal and Agrawal they compared two methods of scheduling socket buffers to determine which resulted in better web server response times. The first method was the standard referred to as FAIR which is based on conventional processor sharing on a first-in first-out (FIFO) basis. The second was the Shortest-Remaining-Processing-Time (SRPT) method in which the server always processes the shortest jobs first. They called this the *unfair* scheduling method. A concise

way to define SRPT is “a greedy strategy to minimize the number of jobs in the system by always working on the job that is closest to completion” (Harchol-Balter & Schroeder, 2006).

The surprising result of this group of researchers was that favoring smaller sized tasks over larger ones resulted in faster response times, in general, even for large or long jobs. They showed that in *unfair* scheduling, larger jobs were either not penalized or only minimally penalized. Furthermore, their technique resulted in no loss to the resulting throughput. Finally, they note that their SRPT concepts can be applied to any resource that is a bottleneck such as the CPU instead of the network, depending on which is the greater bottleneck in a given scenario. They did, in fact, do this at the database level and this is described later. (Harchol-Balter, Schroeder, Bansal & Agrawal, 2003).

However, the work of these researchers, like Abdelzaher and Bhatti, was limited in scope to static requests. Furthermore, they did not consider the case of server overload. (Harchol-Balter, Schroeder, Bansal & Agrawal, 2003). Rather, the case of SRPT during overload was addressed in a separate paper by Schroeder and Harchol-Balter (Schroeder and Harchol-Balter, 2006). While previous research dealt strictly with using SRPT for static content, their new contribution was using SRPT in the context of server overload. Using their implementation of SRPT, they were again able to demonstrate its superiority to the conventional FAIR method of scheduling during an overload condition (Harchol-Balter & Schroeder, 2006).

However, a drawback concerning SRPT should be noted here which is that ordering these requests requires an estimation of the job size in advance. This is referred to as size-based scheduling. While it may be easily achievable with non-dynamic static pages, it is

far more difficult for dynamic content. Even Harchol-Balter and Schroeder (2006) admit that SRPT was previously suggested only for static content. However, they justify the importance of their contribution by noting that the vast majority of web requests are for static content. Furthermore, many sites cache their dynamic content as static content in order to improve processing times. (Harchol-Balter and Schroeder, 2006).

In criticism of SRPT, Cherkasova (1998) notes that always preferring the shortest jobs can lead to *starvation*, or denial of service, to large jobs if shorter jobs continually arrive (Cherkasova, 1998). Instead, she advocates *alpha scheduling with no pre-emption*. In this scenario, instead of prioritizing requests only based on job size, a combination of both time and job size are used. She combines these two values and associates that value with each job. Thus, large jobs are eventually serviced no matter what based on the time in which they were submitted to the queue.

Harchol-Balter and Schroeder (2006), mentioned above, did not address the issue of dynamic content since it is largely related to the database tier instead of the web server. However, in order to achieve performance gains for dynamic content as well, they applied the queuing concepts of their work on the database tier in a different research undertaking. This is because they made their algorithms general enough to be applicable outside the scope of web servers. Just as they modified scheduling of requests to the web server, they also modified the internal database management system lock queues using similar scheduling policies. As in their other research, significant performance gains were achieved by modifying the scheduling behavior of the queue (McWherter, Schroeder, Ailamaki & Harchol-Balter, 2004).

However, a common difficulty with the previous research mentioned is that many of the techniques mentioned are not feasible for average web application developers who typically do not intervene with the code of the systems upon which their applications are built. The implementation of the above methods requires detailed knowledge of the operating system, web server or DBMS architecture. Instead, modifying the application itself is much easier and within grasp of the common developer or organization as a whole.

Besides the research mentioned, there are other well-known techniques for boosting application performance. Casteel (2008) mentioned pushing out processing logic to different layers of the application that are less restrictive. For example, if the maximum memory for server-side scripting is too low, the developer can consider shifting some logic to the database or client-side. This can significantly reduce the load (Casteel, p. 4, 2008). The reason is the reduction in network communication and inter-process communication (Advantages of Using PL/pgSQL, 2010). Also, queries do not have to be re-parsed upon every call. Furthermore, hosting accounts typically do not specify constraints with high specificity regarding the database.

A common example of this is placing logic in stored procedures within the database. This logic could also be placed within the business tier. However, this adds the processing cost of network traffic between the web server and database server as well as the need to parse SQL more frequently. A stored procedure, on the other hand, can remain in a pre-compiled state which avoids these resource expenses (Casteel, p. 4, 2008).

However, placing business tier code in the database, while resulting in better performance, is not always desirable. One disadvantage is that code sometimes becomes less maintainable since the convenience of maintaining all code from a single IDE for one language could be lost. In fact, some frameworks, such as Ruby on Rails, reject this completely despite its advantages (DeVries, Naberezny, p. 89, 2008).

2.2 Queue Theory and Terminology

Researchers Bar-Noy et al. (2009) provide a useful vocabulary for batch processing. A given task is called a job or client. A processing machine is a channel. Each task is given a weighting called the revenue. They define throughput as the number of completed jobs. Table 1 shows a side by side comparison of their vocabulary with the terminology used in this research. (Bar-Noy et al., 2009).

Table 1 Bar-Noy et al. Queue Terminology (Bar-Noy et al., 2009)

Bar-Noy	Current Work
job or client	job
channel	server
weighting termed a revenue	priority
batched	queued

Sundell and Tsigas (2005) also contribute relevant terms related to priority queues. They state that in priority queue theory, the queue is considered a set of *key-value* pairs where the key is the priority of the item (Sundell & Tsigas, 2005).

Slothouber (1996) provided a simple overview of queue theory in relation to web servers in which he elucidates several important concepts. The major contribution of Slothouber's work was a high-level model of a web server as a network of queues. In doing so, however, he also presented many details on queue theory. He stated that every queue has an associated *arrival rate* (A) which is the rate at which new jobs arrive. The

service time (T_s) is the average time needed to process each job. The average time spent in a queue is (T_q) and is called the queuing time. Therefore, the average response time T can be calculated as follows:

$$T = T_s + T_q$$

The *service rate* of the queue is the inverse of the service time ($1/T_s$). A queue system is said to be *stable* if the arrival rate is less than the service rate. In this case all jobs are being serviced and there is an upper bound to the queue size. However, if the opposite is true, the queue is unstable and grows without bound.

Another important metric is utilization of the server which is the product of the arrival rate and service time. A utilization value between 0 and 1 is considered stable while a value of 0 denotes an idle server. If the utilization equals 1 the server is being used to maximum capacity. Slothouber (1996) also discussed Little's Law. It states that, on average, the number of jobs waiting in the queue equals product of the arrival rate and response time. (Slothouber, 1996).

The concept of parallelization can potentially be used for the goal of this research. Parallelization is the running of multiple processes simultaneously. This reduces the load generated by a single process doing the job of all of them. However, it is not yet clear if and how parallelization can be used to bypass the limits imposed by hosting accounts. Snyder and Southwell (2005) discuss its use and benefits in the area of queue processing. The queue can be processed faster if multiple identical processes are handling it. (Snyder and Southwell, 2005).

2.3 Queue Architecture

Literary evidence suggests that queues can be implemented in a number of different ways. Snyder and Southwell (2005) discussed queue usage to control resource consumption. They spoke from the perspective of viewing high resource usage as being a security threat. They went on to describe multiple possible implementations of both queues and the batch processors that handle the queues. Examples of queues they mentioned are a simple directory that stores jobs, an IMAP server or a database (Snyder and Southwell, 2005). Maclennen (2001) covered the basics of a database as an email queue and suggested some enhancements. For example, the body of an email message can be stored in a file whose path is stored in the database. Also, the success and failure of each mailing attempt can be stored in a status field for each message. (Maclennen, 2001).

Herrington (2006) described two alternatives for implementing a queue (Herrington, 2006). He mentioned using a file directory for simple queues and a database table for more complicated needs. Furthermore, he presented some server-side code for both a mail-only queue and a more generic queue suitable for any task. In the case of the generic queue, he suggested storing interpretable code within the database itself in order to provide an action to take for each queued item. However, the intention of the included information was to be a very rudimentary introduction to what is required to design and utilize a queue. It was far from being a detailed design on which to base a commercial system. (Herrington, 2006).

Jim Gray of Microsoft Corporation, a strong proponent of using databases for queue implementation, went as far as to say in a position paper that queues, in the context

of Message Oriented Middleware, *are* essentially databases (Gray, 1995). He argued that since queues need security, configuration, performance, monitoring, recovery, and reorganization utilities, database management systems are the ideal mechanism for implementing queues since they already provide these services. He goes on to say in his section subtitled *Queues are "Interesting" Databases*:

> Storing queues in a database has considerable appeal. The idea is that queues are a database class encapsulated with *create*(), *enqueue*(), *dequeue*(), *poll*(), and *destroy*() methods. By using a database, the queue manager becomes a naive [*sic*] resource manger [*sic*] with no special code for startup, shutdown, checkpoint, commit, query, security, or utilities. Rather it is just a simple application – the database system does all the hard stuff like locking, logging, access paths, recovery, utilities, security, performance monitoring, and so on. The queue manager benefits from all the database utilities to query, backup, restore, reorganize, and replicate data. In addition it piggybacks on the TP-lite and trigger mechanisms of the database system for process and server pool management.

However, he also mentions the difficult problems associated with using databases as queues. For example, enqueue operations require inserts followed by commits which places considerable performance demands on the system because of concurrency control and recovery component operations. Also, deletions can be very complex since they typically involve "deleting a record, processing the request, enqueuing results in other queues, and then committing" (Gray, 1995, p. 6). This indicates the need for specialized isolation levels for queues. (Gray, 1995, p. 6).

Much literature has also addressed processes or systems that coincide with queues. Zhang and Ferrari (1993) contributed research regarding queue architecture for network switches. Nevertheless, some concepts of their discussion are relevant in the context of a task processing queue. Namely, they mentioned the benefits of having a rate-controller unit that regulates the rate of processing a queue. (Zhang, Ferrari, 1993).

2.4 Queue Processing Algorithms

Another area rich with queue-related research is processing algorithms. One of the most useful works was that of Ronngren and Ayani (1997) which carried out an extensive comparative study of several well known priority queue processing algorithms. These algorithms spanned across three main access patterns known as the *Classic Hold*, *Markov Hold*, and *Up/Down*. The researchers kept their scope broad and applicable to many fields. However, the study was in the context of using an in-memory data structure as a queue rather than a persistent queue such as the file system or a database. Nevertheless, many points they discussed are relevant to queues in general. For instance, they discuss the implications of parallel access. Also, they discuss how different factors can affect performance such as queue size (number of elements in the queue) and access patterns. (Ronngren and Ayani, 1997).

Concurrency is another algorithmic issue addressed by algorithmic research. Queues in which access by multiple clients are expected to simultaneously access resources are called *concurrent queues*. Concurrent queues pose a special set of algorithmic concerns addressed in research papers such as that of Sundell and Tsigas (2005). In their paper, they shed light on the issue of keeping a shared data structure consistent in a concurrent environment. The most common and straightforward method is

mutual exclusion which requires *locking* the shared object during access. However, they, among others, have proposed a non-locking algorithm that still achieves consistency. (Sundell & Tsigas, 2005).

2.5 Generic, Reusable Queues

The benefits of a queue are easily understandable. A large workload can be split up over time. However, each application has specific queue requirements. Therefore, it is common in these situations for development to require re-engineering for each specific application (Message-Oriented Middleware, 2010). Thus, having a generic queue that can be re-used is highly desirable since it saves all of the re-engineering effort. One example is where this concept has been implemented because of its great significance is Messaging Oriented Middleware (MOM) such as Oracle's Advanced Queuing. In fact, many parallels can be drawn between MOM and the queue design proposed in this research. MOM stores messages in a queue for client applications to retrieve them at a later time which provides an asynchronous form of communication. Oracle's Advanced Queuing uses a database for persistence, as does the system described in this research. It is used primarily to allow heterogeneous systems to communicate through a standard system. In other words, it provides a generic mechanism for storing data that must be transferred between applications which can be adopted for almost any use. Similarly, the design contained herein details a generic queue can that be re-used by any differing components of a single application. While Advanced Queuing is focused on generic inter-system queuing, this research is focused on generic intra-system queuing such that very different parts of a single application can share a single reusable queue system within it. (Oracle9i Application Developer's Guide - Advanced Queuing, 2002).

2.6 Scheduling

Much research has been conducted in various technical fields regarding optimal performance for scheduling algorithms. The goal of these efforts was to make queue or system scheduling more efficient in general or for particular cases. Other researchers sought to provide a framework such as Marchetti and Cerda (2008) who described a mathematical framework for batch scheduling in resource constrained environments. However, their research was mainly geared towards industrial manufacturing.

One example of more general scheduling work was that of Capannini et al. (2007) where they proposed a method called *Convergent Scheduling* and compared it to other methods such as *Back Filling* and *Earliest Deadline First* (Capannini, Baraglia, Puppin, Ricci & Pasquali, 2007). The resulting product of their work was a scheduling framework. The research they undertook demonstrated how heuristics could be used to result in better and more efficient scheduling. In another study, they designed a two-level scheduler consisting of interconnected sub-clusters for large-scale processing (Pasquali, Baraglia, Capannini, Ricci & Laforenza, 2008).

However, the work of Capannini et al. and most of the current literature is focused mainly on processing items in the queue in the fastest time possible and is not concerned with resource consumption and constraints. This is because it is mainly geared towards large organizations whose application hosting resources are typically not highly restricted. This is immediately evident since the title of their first paper mentioned above refers to ‘Large Computing Farms’. The title of the second one refers to ‘Large-Scale Grids’.

One of the research efforts in throughput maximization was that of Bar-Noy et. al (2009). They made their work purposely general in order to be applicable to anything related to scheduling with batching (Bar-Noy et. al, 2009). In fact, they open by saying the work is usable in fields as diverse as multimedia on-demand and integrated circuit manufacturing. They showed that, while scheduling algorithms typically prevent multiple jobs on the same machine at the same time, this need not be a constraint. Rather, they prove that simultaneous jobs can actually be beneficial. Their work provides great insights into the implications of scheduling and batching.

Another essential component of scheduling is using an external system to run application activities according to a schedule. For example, the Unix *crond* service can be configured to invoke a script every minute, hour or week. Potential queue processors mentioned by Snyder and Southwell (2005) are custom Unix daemons and the *cron* facility. These tools can be used to specify exactly when application execution should be periodically invoked (Snyder and Southwell, 2005). Maclennen (2001) describes a Windows based alternative which is to run the queue handler as a service (Maclennen, 2001).

Regarding process scheduling, Herrington (2006) makes an important argument for choosing a processing daemon versus an application thread. He notes that threads constantly hold memory and could stall or end up in a never ending loop. Using an external process scheduler is much safer from these possibilities. The processes are only run at specified times thereby conserving memory. This is especially important to consider in a resource constrained hosting environment.

Berman, Wolski, Figueira, Schopf and Shao (1996) mention a related extension of scheduling which is *application-level scheduling*. In this scenario, multiple applications access shared resources through a single point of access that must schedule application requests. They refer to important scheduling principles that are relevant to this research. One such principle is that "Dynamic information is necessary to determine system state" (p. 3). In other words, scheduling is optimal when the scheduler has some indication of the current load on the system and the current resources available. For example, if the scheduler detects that system usage is currently low, it can be less restrictive in granting resources or processing tasks. They also maintain that "Good schedules involve some prediction of application and system performance" (p. 3). Berman et al. explain that this principle indicates that having knowledge in advance of expected system behavior provides the ability for the system to choose the best course of action to achieve optimal performance. For example, a scheduler can be configured to increase its activity during times when system usage is expected to be low or when resources are expected to be free.

In order to classify a strategy for scheduling tasks, this researcher referred to the work of Blythe et al. (2005) which relates to an innovative task scheduling strategy in grid environments. They explain that there are two main approaches to scheduling resource allocation to tasks. *Task-based allocation* makes resource decisions based only on the jobs or tasks available at a given instant. However the *workflow-based* approach considers all possible jobs regardless of the moment in time and maps resources in advance. (Blythe et al., 2005).

2.7 Quality of Service (QoS)

The *quality of service* (QoS) of a system has been defined as the "set of quantifiable quality properties of a service" (Enabling a Web of Billions of Services, Glossary). The concept of QoS is important in relation to scheduling and queues since the scheduling system must attempt to guarantee the execution of a queued task before a deadline.

Bhoj, Ramanathan and Singhal (2000) worked on bringing the concept of QoS to web servers. They designed and developed a custom QoS-enabled web server which they called Web2K. This server considered different users as being from different classes and used this to prioritize their requests. Web2K used this as a basis to deliver QoS. Thus, instead of blindly denying service to random requests, under overload the server was able to selectively deny requests. In this manner, the server can provide better QoS for mission critical services. (Bhoj, Ramanathan & Singhal, 2000).

In their paper they mention several important points related to queues. Firstly, one way to attempt a guaranteed response time is to perform admission control. An *acceptor* mechanism checks remaining queue capacity and accepts or denies requests based on the space remaining in the queue. (Bhoj, Ramanathan & Singhal, 2000).

An innovative scheduling queue concept used by Web2K was its temporary storage of denied requests. During overload periods, it temporarily stored these requests and later processed them as higher priority requests. In this manner it was able to easily accommodate bursty periods of traffic. Thus its queue architecture was made more reliable resulting in better scheduling and QoS. (Bhoj, Ramanathan & Singhal, 2000).

Another important contribution of the research team was the demonstration of how queue theory from a certain fields can be successfully applied to a vastly different field. A novel aspect of Web2K was its acceptor mechanism built for a web server while having its roots in techniques used for manufacturing queues. (Bhoj, Ramanathan & Singhal, 2000).

2.8 Conclusion

The literature review process was crucial in developing and refining this research endeavor. It was instrumental in discovering areas still left unexplored. Some issues not covered in the research mentioned above are discussed below.

Furthermore, in contrast to much of the previous scheduling queue work such as that of Capinni et al. (2008) which was directed towards large, powerful information technology infrastructures with massive resources, this current research project aims to develop a system to process a queue based on available but highly constrained resources. This situation is typical of smaller organizations or small departments within a large company or other environment. In such scenarios it is more likely that the management seeks to avoid high hosting costs and maintaining an in-house information technology support department.

Similarly, while most queue scheduling research focused on achieving maximum performance, it seems that research has largely ignored another concern which is achieving *acceptable* performance while giving high priority to the conservation of resources. This can be extremely important especially when the queue is used for non-critical operations. Previous research sought to maximize throughput by controlling and optimizing performance by various techniques such as heuristics (as in Capinni et al.), as

mentioned above. Many of these research efforts do not make any mention of resource limits.

The focus in this paper is quite different. The goal is to process the queue as quickly as possible without exceeding resource constraints or limits imposed by hosting accounts and without causing a significant reduction in website response time. The queue scheduling is to be controlled and QoS maintained by means of user-configured profiles that map performance parameters to periods of time. In other words, any given time has a desired performance profile and the queue must conform to that profile within this time period.

Previous publicly available research has also not provided a detailed design of a persistent queue-based scheduling system. While Gray did introduce some design concerns, his intent, as mentioned, was only to defend a position. Although he mentioned the same general idea being developed in this paper of manipulating a queue implemented as a database with wrapper classes, he did not go into a great level of detail. He also did not expound upon *how* to implement the queue as a database. Similarly, he did not elaborate on the implementation of the queue wrapper classes. Furthermore, he limited his discussion to the scope of MOM.

In this paper, the goal is not only to provide such a research-verified design but to go farther by aiming for a totally generic queue container that is usable by any application for any task no matter how unique it is or how complex its processing requirements are.

While the approach of Abdelzaher and Bhatti (1999) was to make the server respond with lower grade content, this paper focuses on returning the same level and quality of

content and instead optimizing what occurs at the server end. The approach of this paper discusses splitting a request into segments and then queuing those segments to be processed over time. Although not always desirable, it can result in a dramatic performance enhancement in many situations in which immediate processing of large jobs is not required. An additional constraint imposed by the work of Abdelzahir and Bhatti (1999) was that it required content to be pre-processed, which is useless for dynamic content.

Chapter 3 - Methodology

3.1 Background

The nature of this research was qualitative. More specifically, this research began as a constructivist effort in its ontology. This is because no existing software solution was found that could cover provide for the custom needs of the university in which the study was conducted. Instead, a solution had to be created.

Action research was also considered since it stresses iteratively solving a problem through experimentation and adaptation. It was very appropriate for the scenario of making repeated attempts to process massive volumes of queued tasks within a given period of time while continually studying and modifying the system design to bring it closer to the goal.

Shortly into the research, a related but more applicable and structured methodology was adopted, which is *design science research in information systems* (DSRIS). It is a specific subset of constructive research (Kuechler, 2009). It also inlcudes the iterative benefits of action research. In fact, action research can be a component of design science research (Vaishnav and Kuechler, p. 49, 2008). It stresses attaining knowledge through a design-build-evaluate cycle. Its defining characteristic is learning by building. Thus, it was a perfect fit and was taken as the overall design framework of this effort. Vaishnavi's book 'Design Science Research Methods and Patterns' was the core reference used to structure this research.

Design science research works well in an IS development environment by formalizing the development of a prototype and its subsequent evaluation. DSRIS also

has a unique capabilitty for iterative learning and refinement of the research question (Vaishnavi and Kuechler, p 42, 2008).

Design science is also referred to as improvement research (Vaishnavi and Kuechler, p 46, 2008). As opposed to research which aims to explain, it aims instead to "produce and apply knowledge of tasks or situations in order to create effective artifacts" (March and Smith, 1995).

Using DSRIS as a framework, the major steps of the research can be summarized as follows:

- Build a strong awareness of the problem.
- Sugggest a tentative design.
- Implement the design as an artifact.
- Evaluate the artifact against criteria.
- Derive conclusions and knowledge based on the evaluation.

The above steps were taken from Vaishnav and Kuechler (2008) and are called the General Design Cycle (GDC).

Regarding the implementation step, a few points must be noted. The artifact in DSRIS can be abstract in nature such as a construct, model or method (March and Smith, 1995 – Vaishnavi, 49). Furthermore, the instantiation of the artifact can be quite rudimentary since the main purpose is to focus on design and not the actual implementation (Vaishnavi and Kuechler, 2004 – Vaishnavi, 49). March and Smith indicate that conceptual models can be an aritifact of design science (March and Smith, 1995). In this case, a model was made along with an actual software instantiation of.

Evaluation in design science research occurs by using empirical methods “to determine how well an artifact works” (Hevner et al., 2004). Vaishnavi and Kuechler mention multiple evaluation techniques, among which is simulation (Vaishnav and Kuechler, p. 49, 2008). Evaluation can take on multiple forms such as “action research, controlled experiment, simulation, or scenarios” (Vaishnavi, 2004). Hence, the goal is to evaluate the artifact’s utility (Vaishnavi and Kuechler, 49, 2008).

Furthermore, DSRIS stages 1-4 shown in Table 2 are indefinitely iterative such that after exiting from any stage, the researcher can return to any previous stage (Vaishnav and Kuechler, p. 59, 2008).

Another concept taken from (Vaishnav and Kuechler, 2008) is that of research patterns. These authors provide several formal, established patterns to apply to the five steps of DSRIS, or the GDC.

For the problem selection and development stages, the ‘Being Visionary’ pattern was used. This pattern is to “envision an improvement in a situation or problem even if the present sioltion is acceptable” (Vaishnav and Kuechler, p. 95, 2008). In this case the system was working fine in its production environment but the vision of a generic and profile based queue seemed to have several potential enhancing benefits.

For the suggestion and development stages the ‘Empirical Refinement’ pattern was used. Its intent is to “develop a soltuion to the research problem through iterations of system evelopment,empiracle observation, and refinement” (Vaishnav and Kuechler, p. 129, 2008). Table 2 shows the stages of Empirical Refinement (Vaishnav and Kuechler, 2008).

Table 2 Stages of Empirical Refinement

Stage	Title
1	Construct a Conceptual Framework
2	Develop a System Architecture
3	Analyze and Design the System
4	Build the Prototype System
5	Observe and Evaluate the System

Again, these stages are iterative and allow for movement between them. Complex systems are rarely understood completely prior to their implementation (Nicholas, J., 2004, p. 129). During their implementation, knowledge is acquired about how to move the implemenation forward and about the requirements for successfully completing it. In this research effort, the implementation and evaluation of the generic queue system began with a very basic skeletal design. This was later expanded as missing needs and features were encountered and as evaluation provided guidance about missing system requirements.

3.2 Ontology

The ontology of design research recognizes “multiple, contextually situated alternative world-states” and is socio-technologically enabled (Vaishnavi, V. 2008). Thus, it was consistent with the fact that multiple solutions existed for the research problem under focus.

3.3 Epistemology

Vaishnavi states that design research epistemology stresses “knowing through making” an “objectively constrained construction within a context” (Vaishnav and Kuechler, 2008). It also holds the view that “iterative circumscription reveals meaning”.

In this research effort, an artifact was being developed and knowledge about generic queue development and processing was being sought though its construction.

3.4 Methodology

According to Vaishnavi, design research methodolgy is developmental and measures artifactual impacts on the composite system (Vaishnavi, 2008). In this case, the system was developed incrementally until all system needs were accounted for. In fact, the system was developed enough to be used as a production system and it was used as such.

3.5 Research Steps

The steps of this research relied on those given by Vaishnav and Kuechler in their mapping of design research phases to what they call the General Design Cycle. This is detailed in Table 3 (Vaishnav and Kuechler, 2008, p. 59).

Table 3 Mapping of Design Research Phases to the General Design Cycle

Step	General Design Cycle (GDC) Steps	Research Phases
1	Awareness of Problem	Problem definition
2	Suggestion	Literature review, Tentative Design
3	Development	Prototype Development (Artifact Implementation)
4	Evaluation	Simulation
5	Conclusion	Conclusion

The specific steps taken during this research are explained below in greater detail.

3.5.1 Build a strong awareness of the problem.

Queue scheduling and processing is an expansive topic. Thus, one of the most difficult steps was determining in concrete terms what exactly this research was trying to achieive. However, it eventually became clear that the items of greatest interest to the

university supporting this research were related to resource conservation and reusable queue code. Thus, the central goal was to build a generic queue capable of holding any theoretically queable task. Furthermore, it had to function under resource constraints that could vary from one part of the application to another.

3.5.2 Sugggest a tentative design.

In this stage existing knowledge and the well-defined problem definition are syntheiszed into an artifact made to solve the problem. The design was first carefully thought out. It was subsequently developed through a series of iterations. The intial design was suitable for a single type of queue task without any resource considerations. This initial effort led to many realizations about the requirements of building a queue system and provided a foundation for which to begin designing a generic and extendable system that was profile based. The design was iteratively modified even until the very end of development as new needs were discovered and refinements were made.

More specifically, this design encompasses the following items which elucidate the design:

- High Level Integration Diagram – this diagram gives a high-level view of all the system components and how they interact.
- UML Class Model – this portion describes the business layer classes developed in a scripting language which manipulate and control the queue.
- Database ERD and Table Description – this portion describes the database tables, their relationships and the fields in order to clarify their purpose and justification and how they effectively implement the queue container.

3.5.3 Implement the design as an artifact of the research

In order to verify the effectiveness of the design and to test and refine it, the design was implemented as a prototype. Furthermore, this prototype was developed enough to be used on a production server.

The first implementation, based on the first design explained above, was mainly an exploratory effort to gain a firm understanding of the fundamentals of building a queueing system. This implemenation was used in a production environment. Deficiencies were noted and adjustments were made.

Once the fundamentals and the issues surrounding queue systems were thoroughly understood, the researcher felt ready to explore the more complex issues of building a generic profile-based queue.

Subsequently a second major design effort was made which was followed by a lengthy development period that aimed to fully reach the goals set forth by this research effort.

The second implementation built upon the first by implementing a fully object-oriented generic queue system. It was a core system that was extended and manipulated by task-specific and profile based application components such as messaging and enrollment modules.

As stated, the artifact was fully implmented and was eventually used by a university department to handle resource intensive queable tasks, namely enrollment of students and mass email communications. As such, it was a production-level artifact.

3.5.4 Evaluate the artifact against criteria

Under the design research paradigm, the evaluation phase indicates “the quality of the design process and the design product under development”. In other words, the goal of the evaluation was to demonstrate that the artifact fulfills the design goals.

Vaishnav and Kuechler list several acceptable methods of evaluation for DSRIS (Vaishnav and Kuechler, 2008, p. 159-171). Hevner, March, Park and Ram also list many methods and refer to actual examples of evaluation work for design science research (Hever, A., March, S., Park, J., Ram, S., 2004, p. 18).

The most applicable evaluation methods for this project were the demonstration and benchmarking patterns taken from Vaishnav and Kuechler. The demonstration pattern is most appropriate when demonstrating the solution is itself considered a constribution. In this case this applies since demonstration proves the usability of the design’s architecture and program logic. Vaishnav and Kuechler state that the first step is to construct the solution as a prototype which proves that it is a realizable solution. The second step is to demonstrate that the solution is reasonable for a set of predefined situations. These situations “should be predefined and not created to suit the solution”. Furthermore, they should cover multiple problem variations. The result is that the demonstration shows either inadequacy or efficacy of the solution. (Vaishnav and Kuechler, 2008, p. 160). Thus, this pattern was implemented by constructed the prototype and also putting it into actual production use.

The second evaluation pattern taken from Vaishnav and Kuechler is benchmarking, which is used to rate the performance of a solution (Vaishnav and Kuechler, 2008, p. 167). In this case, it is being used to demonstrate the ability of a queue

based system to process more tasks than an older version of the software being used that lacked a queue to process the same task types. Both versions are being measured against the same benchmark to guage the increase in processing power available through the queue design.

During the evaluation stage, the following key questions were considered:

- Is the queue truly generic in that it is suitable for multiple types of queable tasks that differ in nature?
- Is the queue successfully determining its processing parameters based on time of day, day of week, and task type? In other words, is it a profile-based queue as described in the thesis design?

3.5.5 Derive conclusions and knowledge based on the evaluation.

At this stage conclusions were formulated and documented based on the results of the evaluation.

3.6 Final Outputs

The final outputs and deliverables of this research were:

- A detailed system design for building a generic queue that is processed according to configurable resource constraints.
- A prototype/artifact which implements the generic queue.
- Multiple application clients that effectively utilize the queue.

Chapter 4 - System Analysis and Design

4.1 Overview

This chapter includes the queue design and a detailed description of the prototype, which are the major artifacts of the research.

4.2 Design Goals

It is constructive to first lay out the precise goals of the design which follows below. More specifically, this section focuses on the goals that make this design a unique contribution to queue design.

The design was intended to result in a queue that:

- Is technology-independent.
- Is independent of the application described herein and thus replicable for any application.
- Is highly flexible and extensible such that any conceivably queueable task can be placed on it.
- Can determine its performance directives based on user-configured usage profiles and successfully process its tasks within the resource limits set by these directives.

Finally, it should be noted that the queue design, while being implemented for a web-based application, is suitable for a wider range of applications.

The remainder of this section is split up into two major parts. The first (4.3) explains the design and the second (4.4) describes the instantiated artifact and its features.

4.3 Queue Design

This section describes the queue design developed to meet the goals stated above. Sufficient details have been provided to allow the reader to implement this design for any other system.

4.3.1 Component Architecture

The first aspect of the design is the overall architecture of the system components and how they interact with each other. This is the high-level view of the system components that reveals how they interact with each other.

After several iterations of development and reflection, an effective and flexible layered architecture was developed to achieve the stated goals. The major components are shown in Figure 1 in a layered fashion. Figure 1 shows six major layers:

- OS Scheduler – the scheduling application such as *cron* for Unix.
- Shell Scripts – the shell scripts that execute the program invokers (see below) that are grouped together based execution schedules.
- Application Invokers – the programs written in the web application scripting language that can be invoked at any time by any shell script whose purpose is to load pages.
- Queue Processing Controllers – the controllers containing the business tier logic
- Queue Model – the database handling logic
- Queue Container – the database tables holding the actual queue data

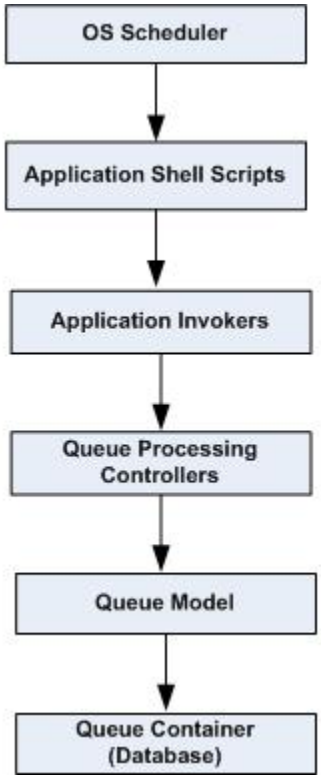

Figure 1 High-Level Component Architecture

It is helpful to consider the *Application Invokers* first. These are server side scripts which call the *Application Controllers* that contain the queue processing logic. These are called in groups by *Application Shell Scripts* based on similar scheduling cycles. For example, all invokers that must be called every minute called at once by a single shell script.

The *OS Scheduler* is not part of the application. Rather, it is an external application utilized to invoke the Application Shell Scripts at specific times.

The Application Controller component contains the actual business logic and utilizes the object classes contained in the *Queue Model* to manipulate the *Queue Container*, which was implemented as a database.

This overall architecture was found to provide maximum flexibility since the scheduler is decoupled from the application by two layers. Furthermore, application invokers are not restricted to a specific periodic execution (weekly, hourly, etc.) since they can be called by any application shell script. Thus, the same application invoker can be called by multiple invokers or its scheduled cycle could easily be modified without having to recode the application.

It should be noted for the benefit of the reader that initial portion of the design, i.e. the layered invocation of the application by the scheduler, is, of course, general to any scheduled application activities and is not limited to queue activity. This same pattern can be replicated for any scheduled activities in any web application.

4.3.2 System Constructs

Marc and Smith (1995) have indicated that design science research efforts produce *constructs* as one of their outputs. Constructs are the conceptual vocabulary of a problem-solution domain (Vaishnav and Kuechler, p. 13, 2008). The constructs of this research are the following:

Generic Queue Core Terms

task – an entry in the queue referring to a unit of work that must be processed and completed.

task type – the type designation of a task. This determines the processing logic for a task.

job – a logical grouping of tasks usually created during the same execution cycle or close together in time. For example, a weekly student report is generated for 10,000 students. These are all submitted to the queue at once as members of the

same job. If all 10,000 reports have been mailed out, that single job has been completed.

task priority – a numerical value indicating the relative priority of a task.

job priority – a numerical value indicating the relative priority of a job.

queue client – Also known as a *Job Processor***,** an application component that makes use of the queue services and contains business logic that processes all of the tasks of a given job. Examples are email messaging and enrollment clients.

profile entry – a directive stored in the database indicating how the queue should behave under a set of environmental circumstances for a given task type. Each entry contains ideal queue throughput values for a given time range and set of days of the week. An example is that during the weekend at any time the queue should send 100 emails per hour. Another profile entry could indicate that during the weekdays from 9 AM to 5 PM the queue should send only 5 per hour.

profile – a set of profile entries for a task type. An example is a profile for messaging that contains two entries. One is for work hours, another is for off hours during weekdays and a third is for the weekend.

active profile – the profile that has been chosen to be used for a given moment in time. The other profiles are *inactive*.

processing cycle – a single execution run of a queue client (see above). Each processing cycle is invoked by the system scheduler.

system scheduler – the mechanism that determines how often and when to run the queue client processing cycles for each job type.

Application Specific Terms

These are terms relating to the specific queue clients developed for the instantiated prototype.

enrollment – the placement of a student into a course section.

enrollment file – a file containing one or more lines with each line containing the data needed to enroll a single student into a section

Each item stored on the queue is a *task* that has a specific *task type*. Example types are *message* and *enrollment* for the system under discussion.

Tasks of a similar nature created at the same time and grouped together logically are assigned to the same *job*. A job can contain any number of tasks. For example, a group of 10,000 email messages all created together for the same purpose all belong to the same job.

A *queue client* can contain multiple *profile entries* in its *profile*. These provide the queue with details about the quantity of tasks to be processed during each *processing cycle*. However, the frequency of processing cycles is determined by the system scheduler.

4.3.3 Queue Container

This section explains in detail the actual software construct that implements the queue storage mechanism in the form of a database.

4.3.3.1 Implementation Overview

Multiple options are available for queue implementation. These have been discussed in the literature review section. In summary, some common options are using a

database, file or an IMAP server. The most suitable option in this scenario for the generic core of the queue is a database. This is because of the ease with which data fields can be stored along with the queued item. For example, an email message has an associated file path for the message body, recipient address, sender address, etc. This is not as easily achievable using the alternatives discussed earlier such as an IMAP server or a simple file directory. Thus, for this particular queue scenario, using a database is the best option.

Another reason to use a database is that this queue should be flexible enough to allow any type of task to be placed on it. This is easily achievable using child tables linked to a parent queue table. The parent table serves as the generic core while the child tables extend it with whatever additional data fields are needed based on the specific queue client needs. Furthermore, views can combine these parent and child tables to effectively produce multiple instances of the queue, each customized with data fields pertaining to a specific application of the queue for a given task type. This is described in further detail during the table and field discussion.

However, it was found during the course of the research that in the application specific queue client layer, relying on only a single queue implementation type for all tasks is not always enough. For example, combining a file queue with a database queue in a multi-queue layered architecture was found to be the most efficient solution for dealing with large numbers of enrollment tasks, as described later. Nevertheless, this is not an aspect of the core generic queue but rather an extension of it in the queue client layer used for an application specific purpose.

4.3.3.2 Database Tables

The design of the queue tables is an essential aspect of this queue implementation. The tables and their fields were carefully chosen and refined to capture the data required to perform the basic queue processing functions as well as to support additional powerful features.

In order to fully explain the design, it is helpful to go table by table and justify the existence and purpose of each table. Before proceeding, it should be noted that the tables were normalized in order to avoid repetition of data.

There are two main groups of tables. The first group comprises the generic core of the queue. This is the portion that can be used in any application, no matter how the queue is being used. The second group is the application specific queue clients. These tables are the child tables that build upon the generic core. Tables in this group naturally differ from application to application.

The entity-relationship diagram (ERD) shown in Figure 3 represents the queue table design. However, additional graphical cues have been added to show the following points:

- The *tasks* table is essentially the heart of the queue since it holds each individual task.
- In addition to the *tasks* table, the tables which form the generic core of the system are the following: *jobs, job_types, task_types, profiles.*
- The *messages, queued_enrollments* and *enrollments* tables are application specific queue clients.

- There is a further dichotomy within the generic core. Tables related to tasks are colored brown while tables related to jobs are colored blue.
- Within the queue clients, *queued_enrollments* and *enrollment_files* are colored grey since they are related. Table *messages* stands on its own.

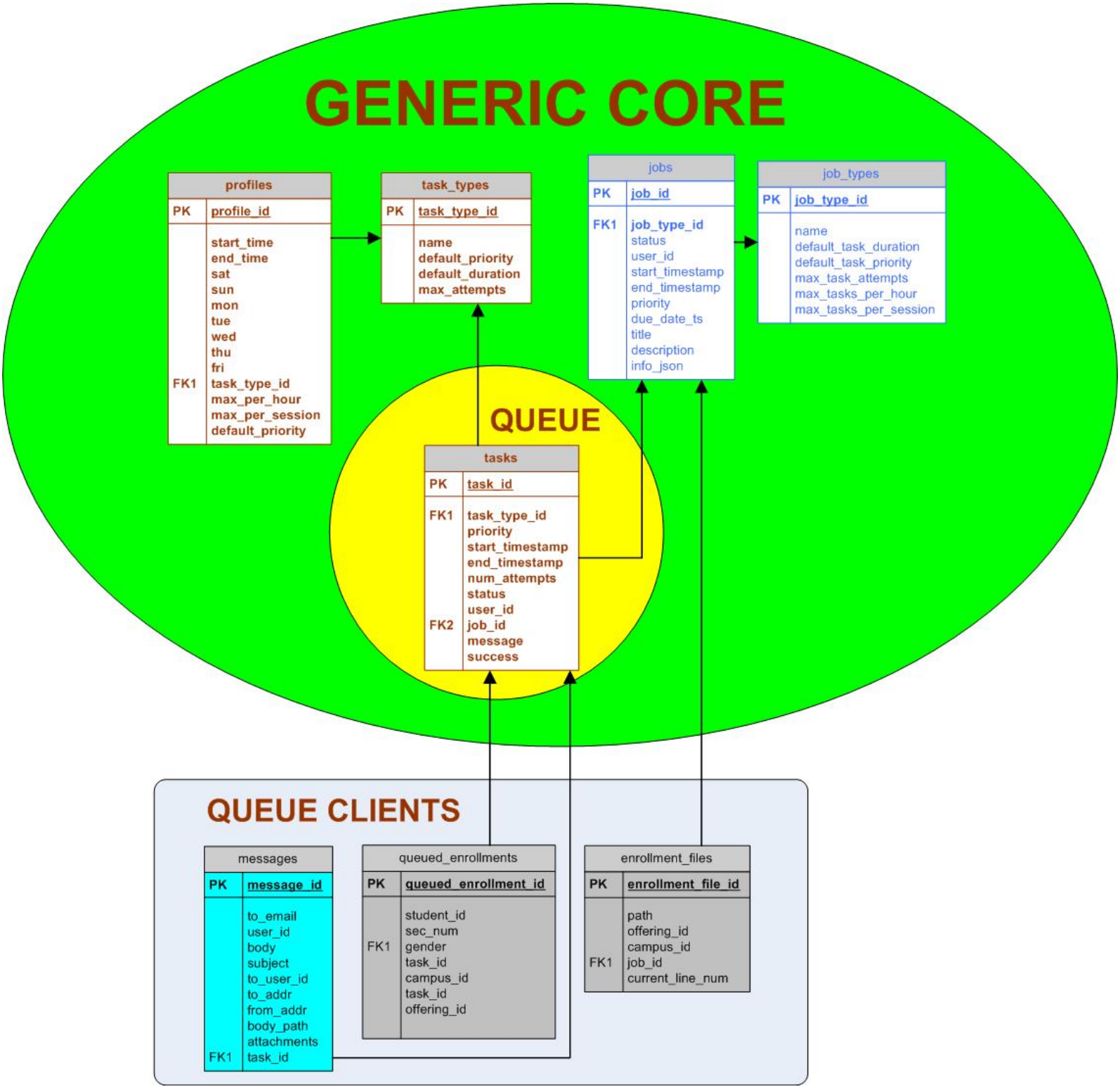

Figure 2 Entity Relationship Diagram for Queue System

Following is a list of the database tables in Figure 2 along with descriptions and justifications of their inclusion.

4.3.3.2.1 Table *task_types*

Each task type has a different nature and requires different processing logic. Thus, table *task_types* is needed to store the different types of queue tasks within the system. A task is the smallest atomic unit processed by the queue. In the prototype developed, three entries existed in this table: *enrollment*, *message* and *enrollment file read*. A brief description of each type follows:

enrollment – the placement of a single student into a section.

message – the sending of a single email

enrollment file read – the reading of a profile-specified number of lines from an enrollment file.

Although not implemented yet, a score import is another potential task type.

A task type cannot comprise more than a single atomic, indivisible unit of action or logic. Joining together different task types is easily achievable using job types, which are described later.

Table 4 shows the fields of table *task_types* and their usage.

Table 4 Table task_types Fields

Field	Purpose	Default Value
task_type_id	Primary key.	
name	Short title/description of the type.	
max_attempts	Maximum number of attempts for the task type.	0
default_priority	Default priority level for items queued as this type.	0

A task should be attempted at most the number of times indicated by *max_attempts.* If it has been exceeded, a task's status should be set as failed.

The *default_priority* field is available to be used in the application to set the priority of a task. However, it can be overridden using the job priority or any application other value.

4.3.3.2.2 Table *tasks*

As explained previously, a task is atomic and is the smallest unit of action in the queue system. This table, whose fields are shown in Table 6, holds the individual tasks that are queued. Examples are a single email message or a single enrollment. This table is the core of the entire generic queue.

Table 5 Table *tasks* Fields

Field	Purpose	Default Value
task_id	Primary key	
task_type_id	Type designation for the task	
job_id	Job type designation of a task.	
start_ts	Timestamp at creation time.	Starting time and date.
deadline	Last possible moment before which to process task.	*start_ts* plus default task type duration
end_ts	Timestamp at end of processing time or at failure time.	
priority	Relative priority.	0
attempts	Number of times this task has been attempted.	0
user_id	Field to track initiating user.	
status	Indication of whether a task has completed or not.	Null
success	Indication of success or failure of task.	Null
message	Field to store any error or notification message associated with the task for later retrieval such as in reports.	

Each task has a mandatory task type designated by *task_type_id* and a mandatory job indicated by *job_id*. When the task is created, it is assigned a timestamp stored in *start_ts*. When it is finished with either success or failure, another timestamp is stored in *end_ts*. *end_ts* is useful in some scenarios, especially for measuring performance of the queue. However, if the application has no need to record when a task was finished, this can be disabled. The default *priority* for the task comes from its task type. However, this can be overridden if needed by the application. When a task is processed, *attempts* is first incremented. If processing is interrupted or unsuccessful, another attempt can be made at any time and *attempts* is again interrupted. This can continue as long as *max_attempts* for the task type is not exceeded.

Two fields are used to track the overall status of the task. *status* indicates whether the task is finished or not while *success* indicates whether the task was logically successful or not. For example, an email message task might have been attempted three times, which is the maximum set for the *message* task type. They all logically failed because of network problems. Thus, *success* is set to *false* because the message was not successfully sent. However, status is set to *done* since the task is no longer being attempted.

An important issue regarding tasks is task processing order. A aspect of this design is that task processing order can flexibly differ at the queue client level. While an initial order can be set in the queue, it can be manipulated later by the queue client. In other words, the application can easily process different types of tasks in different jobs in any order determined by requirements. Even if two jobs use the same task type in their tasks, the order in which tasks are processed can be set differently for the two tasks.

A combination of several task characteristics can be used to determine the default processing order. Tasks are first grouped by their jobs. The jobs themselves are ordered by *priority* and then *deadline*. Within a job, tasks have an internal order. A creation timestamp, stored in *start_ts*, is recorded for each task. Using the *default_duration* for the task's type, a *deadline* timestamp is calculated.

However, *priority* also comes into play since a higher priority task should be processed first even if it has a later *start_ts* value or if another tasks deadline is about to expire (discussed below). Furthermore, the job priority associated with the task's job type can be used to add another layer of ordering. For example, tasks could be ordered by job priority first and then with each job by priority and next by *deadline*.

It is up to the queue client to decide the final task processing order. Since this is a generic queue where multiple tasks are mixed together, the queue clients must look at a subset of the queue to determine order. For example, the messaging queue client must pick the next message by first filtering out message tasks only. The *tasks* table is itself ordered by record insertion time. This filtration along with default ordering of tasks is provided by views discussed later.

4.3.3.2.3 Table jobs

Table *jobs*, explained in Table 7, is used to logically group queue items together. For example, a single mailing might be sent out to 1000 students. These are all added as a single *job* consisting of 1000 different emails that must be sent.

Table 6 Table *jobs* Fields

Field	Purpose	Default Value
job_id	Primary key	
start_ts	Timestamp at creation time.	Current time and date.
end_ts	Timestamp at job end time.	
deadline_ts	Calculated deadline timestamp for job.	*start_ts* plus default job type duration.
priority	Job priority level.	Priority level from job type.
status	Indicates success, failure, pending (default).	Null
name	A user-defined name.	
job_type_id	A type designation for the job.	
info_json	A field containing custom job information stored in Javascript Object Notation.	Null

An important aspect of this design is that jobs are not limited to a single task type. In other words, a single job could involved several different types of tasks. An example could include a job that must 1) enroll a group of students, 2) email each on a welcome letter and 3) email the administrative staff a list of successful imports. In this manner all of these groups of tasks have been logically grouped together within a single entity.

As in with tasks, job are processed in an application determined order at the queue client level with a default order set within the queue. The default job processing order is to consider priority followed by deadline. This is described further in the views section below.

An important feature of this table is the *info_json* field. This field can be populated with any supplementary information that needs to be associated with the job. It is stored in the lightweight Javascript Object Notation (JSON). JSON makes an ideal notation for storing basic data since a large number of libraries are available to parse it and they also provide a number of other features for manipulation (CITE). Its usage can

be made clear through a simple example. One of the jobs in the prototype was an emailer whose role was to add announcement messages to the *message_queue* such that a single message is stored for every student in the queue. The job is executed every minute and adds 5 messages at a time. The mailer job is later responsible for actually emailing these messages by processing the *message_queue*. However, how can the job remember between executions the title of the message to be sent and the path in which its body is stored as a file? This information and any other associated information that is needed between the start time of the job and its end time can be stored in the *info_json* field and retrieved easily every time the job is run.

4.3.3.2.4 Table job_types

Table *job_types* contains the different types of jobs whose tasks can be placed on the queue. It allows default processing parameter values can be associated with different job types.

Table 7 Table job_types Fields

Field	Purpose
job_type_id	Primary key
name	Short title/description of the type
max_tasks_per_hour	Maximum number of items to be processed of this type per hour, this can be altered by program or system administrator.
max_tasks_per_sess	Maximum number of items to process per execution. For example
default_priority	Default priority level for items queued as this type. This can be easily overridden anywhere in the application.
default_duration	The time that should be used to calculate the *deadline*.

4.3.3.2.5 Table messages

This is a table holding information for the messaging queue client. [EXPAND]

4.3.3.2.6 Table enrollments

This is a table holding information for the enrollment queue client. [EXPAND]

4.3.3.2.7 Table profiles

Each task type can have one or more profiles stored in table *profiles*. These profiles indicate throughput parameters for a given combination of time and day of week. The fields are shown below.

Table 8 Table *profiles* Fields

Field	Purpose	Default Value
profile_id	Primary key	
task_type_id	Type of task this profile applies to.	
max_per_hour	Maximum tasks to process per hour.	1
max_per_sess	Maximum tasks to process per session.	1
start_ts	Starting timestamp at creation time.	Current time and date
end_ts	Ending timestamp.	
Sat	True if profile includes Saturday.	false
Sun	True if profile includes Sunday.	false
Mon	True if profile includes Monday.	false
Tue	True if profile includes Tuesday.	false
Wed	True if profile includes Wednesday.	false
Thu	True if profile includes Thursday.	false
Fri	True if profile includes Friday.	false

For example, a given profile for weekend hours might have work days set to false and weekends to true with high values set for *max_per_hour* and *max_per_sess*.

4.3.3.3 Views

In order to facilitate queue processing, database views can be developed to combine important fields from different tables. Furthermore, since this is a generic queue with mixed task types in it, they are especially important to provide filtered versions of it. Finally, the views also provide a convenient default order to the tasks to be used in processing.

4.3.3.3.1 View jobs_ordered

This view displays all fields of the *jobs* table ordered by priority in descending order and then deadline in ascending order. Thus, the highest priority job whose deadline is earliest appears in the first row.

4.3.3.3.2 View tasks_ordered

This view displays all fields of the *tasks* table ordered by priority in descending order and then deadline in ascending order. Thus, the highest priority task whose deadline is earliest appears in the first row.

4.3.3.3.3 View job_tasks

This view combines *jobs_ordered* and *tasks_ordered* to give a final view of all tasks along with related job information for each task. More importantly, it gives an order that is derived from both job and task information. This view is ordered first by the ordering of *jobs_ordered*. Then, within the rows for each job, the ordering is taken from *tasks_ordered*.

4.3.3.3.4 Queue Client Views

Queue client views combine together all tables and fields relevant to a single queue client. The client views build upon the core views. Views can be ordered according to the need of the queue client.

For example, the *message_queue* view provides in each row all information relating to a single message task. It is ordered by job priority, task priority and then deadline. A similar view exists for enrollments.

It is not essential for a queue client to use the default ordering provided by the queue.

4.3.4 Task Scheduling

The scheduling component of the design was developed to allow very flexible and decoupled invocation of the queue clients. Direct invocation of the many web application

components by the scheduler can become very difficult when the number of scheduled tasks is high. In complex applications, the number of entries in the scheduler file could easily become unmanageable. This it is not very portable. If the application were moved to a new server, scheduled tasks could be tedious to migrate because of operating system or scheduler differences. It is better to push all of the references to application logic into the application directory structure itself in a scheduler independent way. This can be achieved by splitting the process of invocation into a series of steps. In this design, scheduling configuration has been completely decoupled from the invocations of program logic. The scheduler is configured with directory names within the application and instructed to execute whatever scripts lie in them regardless of what they are. These directories each contain scripts that must be run on the same schedule. A given directory contains all files that must be run every minute while another contains all script files that must be run once per week. As an example, this web application has multiple processes that must be run on different schedules. For example, report creation may occur once a week whereas emails are processed from the queue once every minute.

The scheduler never invokes the application directly. Instead, it invokes these *scheduled scripts* discussed above which in turn invoke one or more components of the application that need to be run during the current schedule cycle. In this manner application tasks can be reshuffled without having to adjust the scheduler configuration (such as *crontab* files) at all. Instead, only the scheduled scripts need to be modified. Thus, portability is greatly enhanced since changes are required only within the application's directory contents.

The suggested directory structure appears in Fig. 3 as viewed through a file browser. The *web_application* is the document root of the web application. The *scheduling_files* subdirectory contains all files related to scheduling. Within this directory, there are two directories which each hold a different type of script file. The *application_invokers* directory holds *application invokers*, which are files that are used to invoke components of the application but contain no actual application logic themselves. The *scheduled_scripts* are files invoked by the system scheduler written in an operating system command line scripting language. They in turn call the application invokers written in the programming language of the application. In this manner, the application is made to be completely independent of the scheduler. The *scheduled_scripts* directory contains subdirectories that group files based on the similarity of their periodic execution cycles. All files that need to be run every one minute are placed in the *1min* directory. All files that need to be run every hour are placed in the *hourly* directory and so on. However, if greater customization is needed in scheduler configuration, there is nothing preventing the scheduler from calling the application invokers directly.

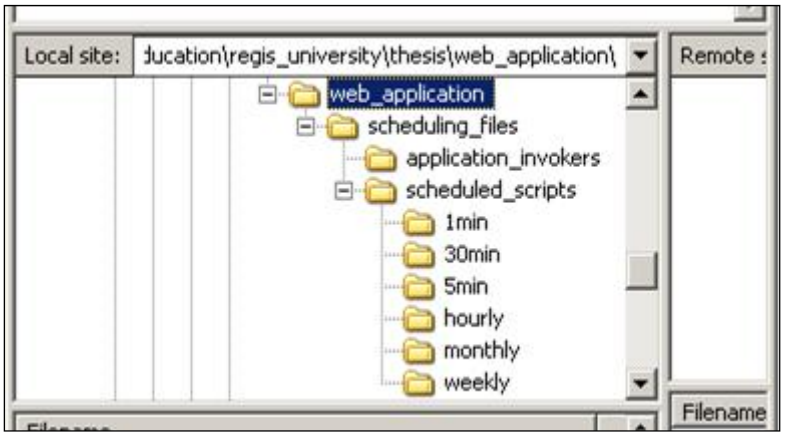

Figure 3 Directory Structure for Scheduled Tasks

Fig. 4 shows an example scheduler configuration using *cron*. The highlighted line shows how just one line is needed to invoke all scripts that need to be run once a week. The same can be done for hourly and monthly tasks.

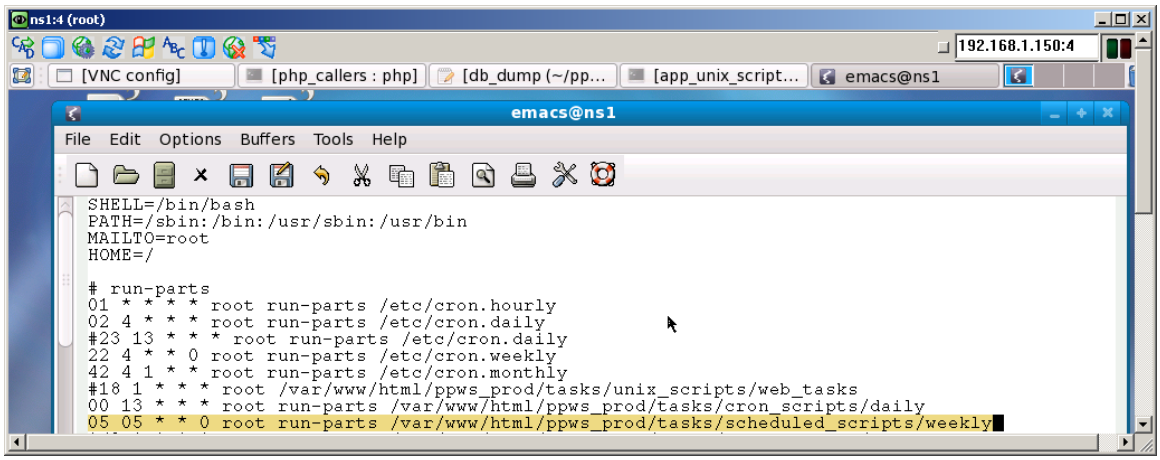

Figure 4 Example Scheduler Configuration

4.3.5 Queue Processor

The processing of the queue takes place in server-side logic. The design of the object-oriented queue processing mechanism consists of classes representing the major aspects of the queue system. The classes are described below and their purposes are justified. Furthermore, the behavior of the processor is discussed and explained.

4.3.5.1 Profiles

The queue client is responsible for choosing the profile to use based application criteria. The profile chosen during an execution cycle is referred to as the *active* profile. In this manner, the queue determines its own behavior. The profile-based action of the queue allows customizable behavior and that is what gives it the ability to react to

changes in its environment. Profiles provide performance and processing directives based on different environmental factors.

Profiles are specific to task types. Thus, messaging and enrollment can have their own individual profiles. A specific set of messaging profiles could be summarized as follows:

Table 9 Example Profile for Messaging

	Weekday Work Hours	Weekday After Hours	Weekend
Profile Values			
start_time	7:00 AM	12:00 AM	
end_time	12:00 AM	7:00 AM	
Sat	*	*	
Sun	*	*	
Mon	*	*	
Tue	*	*	
Wed	*	*	
Thu			*
Fri			*
max_per_hour	5	1000	1000
max_per_session	1	5	5
default_priority	1	1	1

The first profile is applied from Saturday to Wednesday and indicates that during work hours (7 AM to 12 AM), 5 emails can be sent per hour. After work hours, the queue can send 1000 emails per hour since the server needs less resources to devote to users. During the weekend, the number remains at 1000 at all times. Note that Thursday and Friday are the weekend for the university in which the study is taking place.

Upon each execution, the queue client determines which profile should be used based on the current time and day of week. It then retrieves the number of tasks it should process which is indicated in the profile. It keeps doing this unless it exceeds the

maximum tasks per hour set in the profile. Thus, behavior of the system during work hours can be drastically different from non-work hours during which most users are not using the system. The general scenario has been outlined here. However, processing based on profiles can be customized in each queue client if needed.

Profile selection can theoretically be based on many different factors. However, in this scenario the profile was selected based on day of week and time of day. The application's user interface is expected to prevent the entry of overlapping profiles.

It must be noted that the profile selection features and algorithms developed in this prototype were basic yet sufficient for application's requirements. The goal of its development was to be a proof-of-concept. Only day of week and time of day were considered in the profile data. Nevertheless, it provided a tremendous benefit. Furthermore, the profile features and algorithms could be greatly enhanced for both this and other applications to provide further power and customization.

4.3.5.2 Model

The queue is manipulated directly by its corresponding classes. The relationships between these classes are shown in Fig. 5.

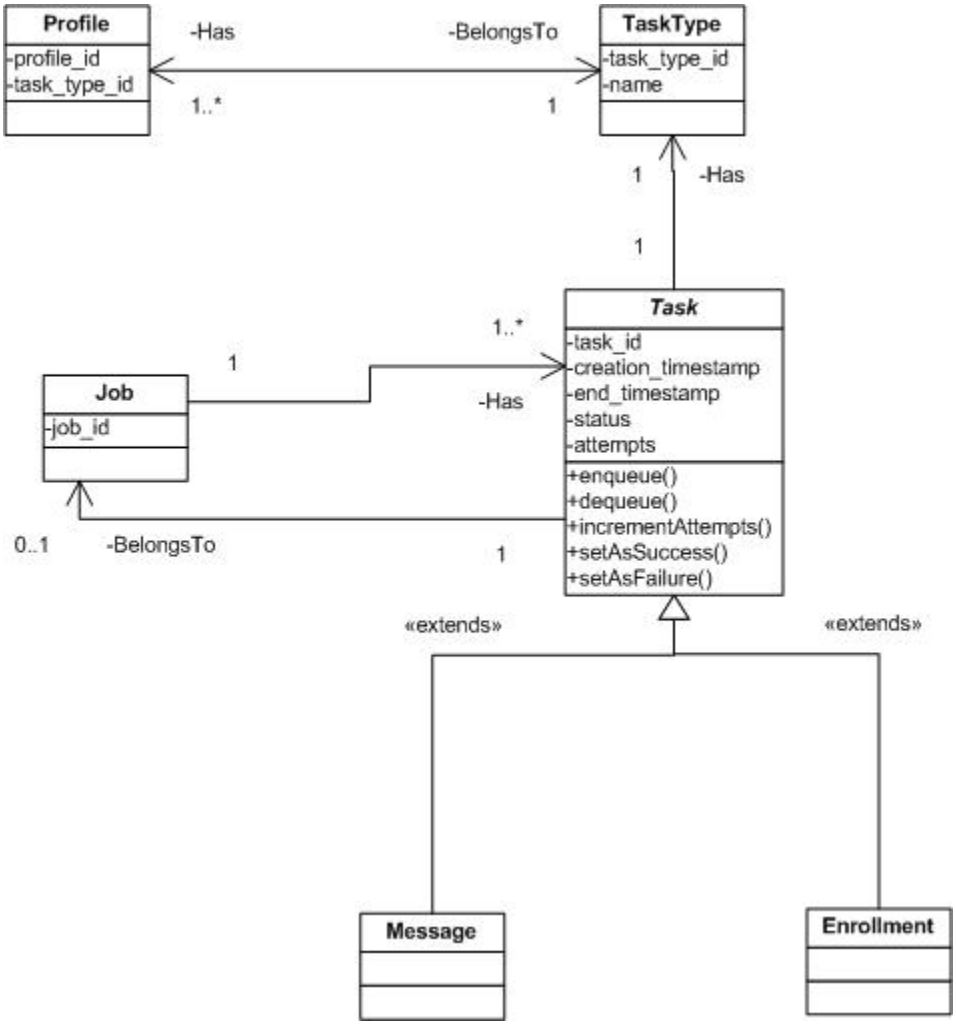

Figure 5 UML Class Diagram of Queue Classes

Task is an abstract class that is extended by the classes representing the queue clients such as **Message** or **Enrollment**. Its attributes are completely useable in the sub-classes. However, its methods must be implemented in the sub-classes since adding a message to the queue requires different logic than adding an enrollment.

Note that there is association between **Job** and ***Task*** indicating that a ***Task*** can belong to at most a single **Job** if any and a **Job** can contain many tasks.

Each ***Task*** requires a **TaskType** object as an attribute. Finally, each **TaskType** object can possess multiple **Profiles** while a **Profile** is unique to a single **TaskType** object.

4.3.5.3 Controllers

The server side controllers contain the logic that manipulates the model. There should be at least one controller per queue client along with controllers for other purposes such as queue analysis and maintenance.

The controllers are invoked by the scheduling system in order to carry out specific actions at specific times.

4.3.5.3.1.1 Mailer

The mailer controller manages messages in the queue. It is executed every minute by the scheduling mechanism. During each execution it first determines which profile is currently active. It then checks the maximum number of emails that can be sent during the current hour and checks whether that maximum has been reached. If it has not been exceeded, pending messages are retrieved to be sent. Processing throughput values can come from three sources. The first is from the task type. The second is the job type.

The number of messages retrieved is determined by the **max_per_session** parameter in the chosen profile.

If a message was sent successfully, the task can then be mark as completed along with corresponding message body files.

A central goal of the system is to always attempt to have emails sent before their deadline. An important issue to consider is what happens if tasks and jobs are behind

their deadlines. If the deadline for a job has passed and emails still remain, other jobs should not be delayed as a result. At this point, the queue system processes the late job and the next items in the queue simultaneously so that the late job is gradually finished off while the other jobs begin. In this scenario, the controller retrieves an equal amount of emails from each remaining job and serves all jobs in a round-robin fashion.

The system tracks how many tasks have been processed in the current hour in order not to exceed the *max_per_hour* directive associated with the messaging task type. This value is saved in a usage statistics file called *queue_stats*.

4.3.5.3.1.2 Enroller

The reason queuing is needed for enrollment is that it requires many integrity checks that consume resources. Performing them at the same time as file parsing was found to exceed typical hosting constraints by a lot. Separating them removes this limit. The goal in using the queue for enrollment was place no restriction on the number of enrollments that can be imported from a file other than the maximum file size that can be uploaded through a browser.

For a small number of enrollments, the user can choose immediate processing. For large enrollment jobs, an imported file is not processed at all. It is simply saved on the server and a job is queue to process it. Processing this queued enrollment file requires two scheduled processes. The first is to parse the files line by line and enqueue the enrollment attempts. The second process is to actually enroll the students in the queue while checking for errors such as lack of prerequisites or incorrect student numbers. If the enrollment file is fully processed, the user is notified by email of success or failure.

4.3.5.3.1.3 Queue Maintenance

Maintenance of the queue refers to deletion of completed tasks along with their associated data and files. For example, if a message has been sent, the file containing its content can be deleted. This can be done immediately after sending or by a scheduled queue maintenance script which checks for all tasks and files that are eligible for deletion and deletes them. The prior solution is more direct and timely.

4.4 Design Instantiation (Prototype)

As noted, the prototype was not a separate software application. Rather, it was developed as a part of an existing university web application in order to demonstrate and test its usability and the benefit of the queue design.

The prototype was developed to a great extent. This is because it was intended and used for production level operations. It served a community of approximately 150 faculty and 5000 students. Processing times were satisfactory. The prototype successfully ran on a VPS shared hosting solution with low resource allocations that were only slightly above the most basic VPS package found.

Chapter 5 - Evaluation

This section includes a formal DSRIS style evaluation along with a discussion of the above analysis and results. As stated above in the research methodology section, evaluation in DSRIS consists of proving the utility of the artifacts produced. The artifacts produced in this research were the full queue design and the instantiation of that design. Two methods are being used to evaluate the artifacts. These methods are defined by Vaishnavi and Kuechler (Vaishnavi and Kuechler, p. 160, 2008). The first method being used is *demonstration*. By demonstrating the functionality of the prototype, the design is proven to be effective as well as the prototype itself. More specifically, in this case the queue is proven to be generic by successfully handling multiple types of queuable tasks. Also, the queue is proven to be configurable and profile-based by successfully incorporating profile values into its operation and demonstrating a difference in behavior resulting from the selection of different profiles. The second method is *benchmarking*. The system is tested and its performance is compared to a non-queue solution.

5.1 Demonstration

This section describes the demonstration-based evaluation of the design through its prototype. It is broken down into different design goals of the system which are each demonstrated in order to prove that they were successfully met.

5.1.1 Generic Nature of Queue

The queue proved to be truly generic in nature. It was used for a wide variety of job types and task types as shown in Tables 9 and 10. All were successfully accommodated by the queue. Countless programming hours were saved since a separate

queue system did not have to be developed for different application components. Rather, the same generic queue core was reused several times.

Furthermore, many of the job types were not preconceived requirements. Rather, the need for them arose after the queue was already developed. Thus, the fact the queue easily handled these new job and task types proved its generic nature. Several actual examples exist. For instance, at one point it was decided that, in addition to emails being sent to all students enrolled in a given course, another need was to simply send out an email to all students in the system no matter what course they were enrolled in. Using the highly flexible, extendable

Table 9 shows the task types used in the prototype. There were three main types.

Table 10 Task Types

Task Type Name	Purpose
Message	Sending an email message
Enrollment	Enrolling a student into a section
Enrollment file	Reading a block of lines from enrollment file and queuing them for enrollment.

Table 10 displays the job types used in the application of which there are currently six. Three of them make use of more than one task type. For example, the enrollment import job performs many enrollment tasks and also queues a new confirmation message task to inform the user about the jobs status.

The table displays various characteristics about each job type which makes it evident that, although composed of the same basic task elements, these jobs differ widely in nature and requirements. Furthermore, since the moment the queue was put into production use, new job types arose regularly and the queue successfully handled them.

The fact that the queue was able to handle all of these jobs successfully is proof of its generic and flexible nature.

Table 11 Job Types

Job Type Name	Average Tasks Per Job	Tasks Per Session	Frequency	Average Duration	Task Type(s)	Description
Enrollment Import	600	10	60/hr	10 min	Enrollment	Import a list of enrollments in a CSV and immediately add them to the enrollment queue. The process the queue gradually. Finally, email an error report to the user who submitted the job.
Weekly Student Report	5000	1	12/hr	5 days	Email	Generate a report for every student enrolled in a course and queue it for mailing. Gradually process the queue.
Student Mailing	5000	1	60	5 days	Email	Queue one copy of an email message for every student. Also send a copy of the email to the user who submitted it. Gradually process the queue.
Enrollment File Processing	600	10	60/hr	30min	Read enrollment file, enrollment	Upload a file containing enrollments to be processed. Gradually process the file. During each session, store the file enrollments in the enrollment queue. Process this queue gradually. Email a report of errors to the user.
Daily Logon Report	1	1	1/day	15 min	Email	Generate a list of users who logged on today and users who did not. Email this list to the academic program supervisor.
Emailing Single Message	1	1	1/min	5 min	Email	A simple job to send a single email message.
Offering Copy	700	10	1/min	24 hours	Enrollment	Gradually copy the enrollment list of one offering to another offering.

5.1.2 Effective Use of Profiles

A specific instance in which the configurable profiles were highly effective and provided a significant, novel advantage over the previous version was that system behavior varied between the development and production application versions without having to change any programming. Programming logic on both servers was exactly the same. However, the profiles were simply adjusted to reflect the fact that the development server was able to process the queue much faster for speedy development work and testing. The production server profile was adjusted for slower processing in order not to interfere with user activity. This was one of the main initial goals of the system and it was achieved.

5.1.3 Completeness, Correctness and Reliability

An indicator that the queue system produced was complete, correct and reliable is that it was used at a production level rather than only in an experimental development environment.

Careful analysis of data stored in the queue showed the following:

- Job id's were being assigned correctly.
- All tasks were being processed. No tasks were being skipped.
- Tasks were receiving a proper status designation of success or failure.

5.1.4 Performance and Timeliness

The queue system demonstrative a massive performance improvement over its previous non-queue based predecessors. This is discussed further in the benchmarking section.

In terms of timeliness, the queue was found to always process its tasks in an acceptable time frame. It successfully met the deadlines assigned to each job and to each individual task.

It was also noted that sending emails typically succeeded on the first try and the system was able to process approximately 10,000 emails per week using a rate of one email per minute with no problem.

5.1.5 Cost Savings

Another demonstration of the efficacy of the prototype is the real-life cost-savings effect it had on the organization that utilized it. One example of an unexpected situation in which the software artifact provided a major cost savings occurred during its production use. The university department grew significantly in student population. A feature to copy an offering's enrollments to another was failing because the VPS resource limits were being exceeded. A successful solution was devised which was to define an offering enrollment copy job. This job's role was to gradually copy enrollments from one offering to another without consuming too many system resources at a time. The number of students to copy every minute was set to a small number. The enrollment job was completed within 24 hours which was tolerable for the department. In the end, the department was saved from having to upgrade to a more expensive virtual private hosting plan.

5.2 Benchmarking

Benchmarking was used in order to present actual data proving the advantage provided by the queue system. The application feature used for benchmarking was enrollment since this is the most resource intensive activity of the system. Two sets of

data are presented below. In the first part, the new queue based system was tested against the old pre-queue method. Both systems were tried to determine their resource usage under successive levels of intensity. Furthermore, the time and memory required for each upload operation was also recorded to measure performance as well. In the second part, data is presented about what was achieved on an actual production virtual private sever using the queue system.

5.2.1 Server Resources and Configuration

This section provides an overview of the resources allocated on both the development and production server. They are displayed in Table 11.

Table 12 Server Resources

Type		Development	Production	Description
Database				
	Shared Buffers	1 MB		
	Temp Buffers	1 MB		
	Work Memory	1 MB		
Server side scripting				
	Max Execution Time	1000 sec	100 sec	
	Max Input Time	1000 sec	100 sec	Parsing request data
	Memory Limit	200 MB	150 MB	Maximum amount of memory a script may consume
Hardware				
	Memory	4 GB	288 MB	
	CPU	1 x Intel Core2 Duo 3.00 GHz	3 x Intel(R) Xeon(TM) CPU 3.20GHz, 2048 KB cache	
Operating System		Windows XP	Cent OS (Linux)	

5.2.2 Direct (Non-Queued) vs. Queued Enrollment

This section shows the results of a crucial test to gauge whether the entire effort produced any advantage over the old system or not. Figure 6 shows a screen shot of the application used to carry out the benchmarking experiment. The system provided feedback indicating the number of successful and unsuccessful enrollments along with the peak memory usage and the time required for execution. Furthermore, the user is

provided a choice to use the direct, unqueued enrollment method or the new queue-based method. In this manner, both systems were compared.

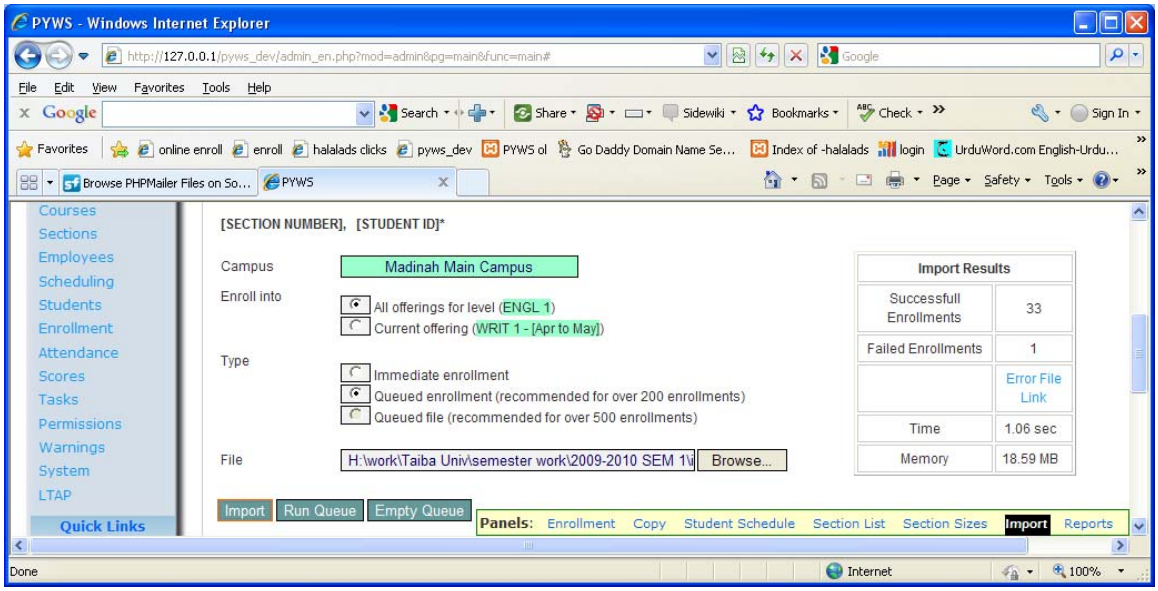

Figure 6 Benchmarking Interface

The enrollment database table was emptied out between every run to give each upload operation the same exact circumstances. The enrollment files uploaded were also optimized to prevent failed enrollments which add execution time because of thrown exceptions.

Table 13 displays the required execution time of the direct, non-queued enrollment versus the new queue based mechanism in order to measure the resulting increase in the number of imports that could be uploaded at once. The table shows the required time to process the file as well as the peak memory consumption of the application while facing successive levels of activity.

Table 13 Required Execution Time

Number of Enrollments	Queued Time (sec)	Direct Time (sec)	Queued Versus Direct
25	**0.33**	**1.66**	**20.00%**
100	**0.99**	**5.00**	**20.00%**
200	**1.52**	**10.12**	**15.00%**
300	**2.22**	**15.17**	**15.00%**
400	**2.93**	**20.44**	**14.00%**
500	**3.69**	**25.79**	**14.00%**
600	**4.46**	**31.41**	**14.00%**
700	**5.29**	**37.04**	**14.00%**
800	**5.82**	**43.74**	**13.00%**
900	**7.03**	**47.55**	**15.00%**
1000	**7.45**	**53.91**	**14.00%**
1500	**10.96**	**86.90**	**13.00%**
2000	**14.76**	**112.68**	**13.00%**
3000	**23.70**	**174.16**	**14.00%**
Average			**14.86%**

The results in Table 13 indicate that the queue method of enrollment takes, on average, only 15% of the time required by the direct method. Of further significance is that the direct method shows that at approximately 600 enrollments it reaches an unacceptable execution time based on an allowed execution time of 30 seconds, which is common in many hosting plans.

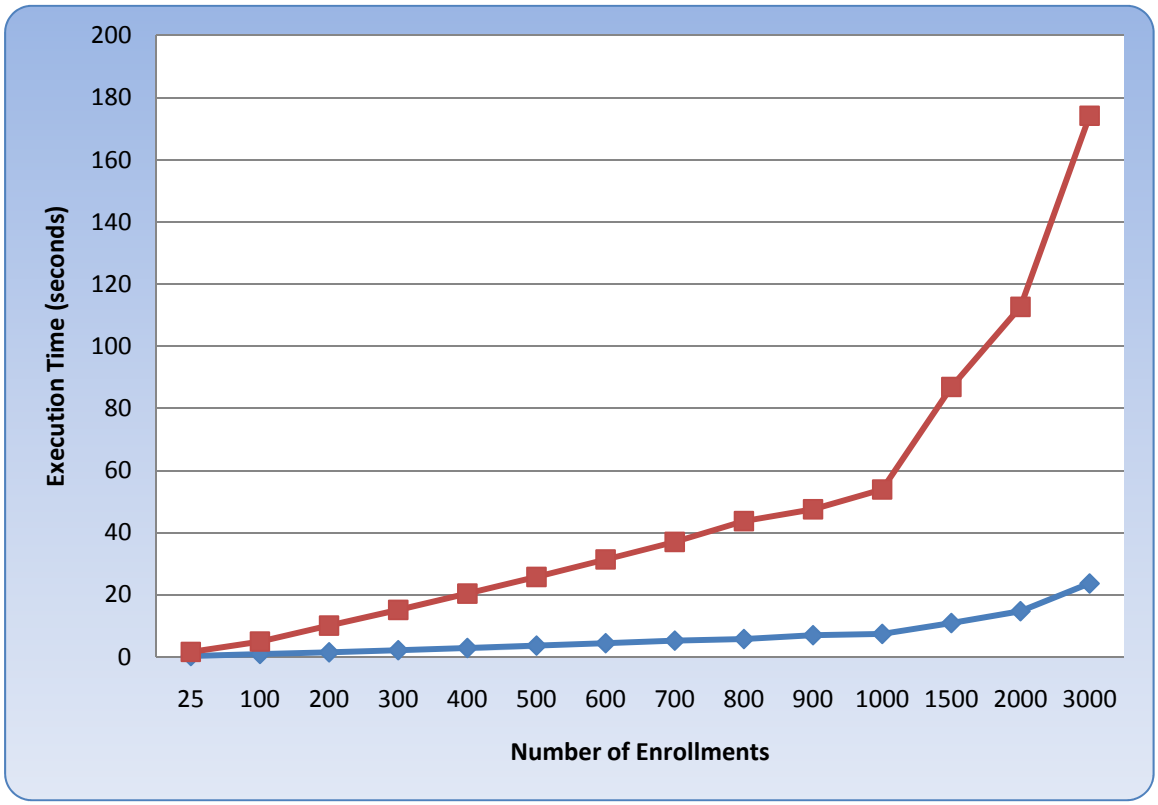

Figure 7 Execution Time Comparison

Figure 7 portrays the data of Table 13 graphically and emphasizes how the improvement in performance becomes even greater as the number of enrollments increases.

Table 14 Enrollments Per Second

Number of Enrollments	Queued	Direct
25	**76**	**15**
100	**101**	**20**
200	**132**	**20**
300	**135**	**20**
400	**137**	**20**
500	**136**	**19**
600	**135**	**19**
700	**132**	**19**
800	**137**	**18**
900	**128**	**19**
1000	**134**	**19**
1500	**137**	**17**
2000	**136**	**18**
3000	**127**	**17**
Average	**127.35**	**18.57**

The data in Table 13 was used to derive an average number of enrollments shown in Table 14. The queue based system achieved a 586% increase over the direct method.

Table 15 Memory Consumption

Number of Enrollments	Queued	Direct	Queued Versus Direct
25	**9.11**	**9.25**	**98.00%**
100	**13.28**	**13.99**	**95.00%**
200	**18.75**	**20.36**	**92.00%**
300	**24.33**	**26.61**	**91.00%**
400	**29.82**	**33.10**	**90.00%**
500	**35.51**	**39.35**	**90.00%**
600	**40.99**	**45.59**	**90.00%**
700	**46.48**	**51.85**	**90.00%**
800	**51.99**	**58.32**	**89.00%**
900	**57.85**	**64.80**	**89.00%**
1000	**63.35**	**70.84**	**89.00%**
1500	**90.78**	**101.42**	**90.00%**
2000	**119.06**	**133.51**	**89.00%**
3000	**173.92**	**195.26**	**89.00%**
3189		**MAX**	
3619	**MAX**		
4000	**FAILED**	**FAILED**	
Average			**90.79%**

Table 15 displays a comparison of memory consumption which turned out to be approximately the same for both systems with the queue system having slightly better

performance. On average, it consumed 90% of what the direct method required. Memory consumption turned out to be the primary limiting factor of the experiment. Both systems failed at 4000 enrollments because they exceeded the maximum allowed memory. However, before failing, the queue system was able to perform 3619 enrollments while the direct method reached only 3189. The consumption is also displayed graphically in Figure 8.

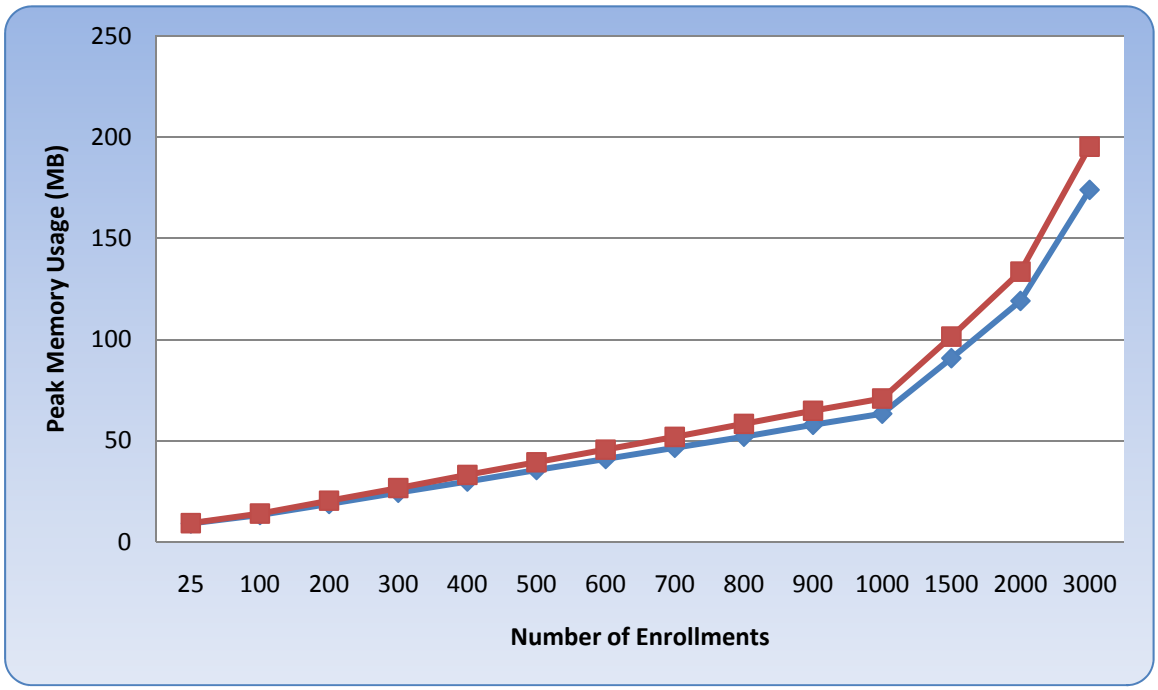

Figure 8 Memory Consumption Comparison

It is important to interpret the data above correctly. The actual execution time, memory consumed and number of enrollments achieved are not highly significant since this differs based on server resources. Rather, it is the percentage change achieved as shown in the tables. Furthermore, the fact that queue system did not achieve a drastic improvement in memory consumption as it did for execution time does not mean that the objectives were not met. Rather, it indicates that the queue system is able to achieve what

the direct method did in far less time which virtually eliminates the possibility of users facing connection timeouts as was common in the older direct method of enrollment. In summary, the queue system can accomplish what the previous system could in only 15% of the time and using 90% of the former memory required.

5.2.3 Queue System Actual Performance in a VPS

The production server underwent similar testing and achieved 2827 maximum queued enrollments compared to 959 maximum direct enrollments. This represents a 195% increase in performance based on the resources allocated.

Chapter 6 - Conclusion

This endeavor was both a learning experience and a knowledge generating process. Its end product was the design for a highly reusable and extendable generic queue core system as well as a prototype implementation.

Furthermore, the queue performance is configurable through the use of profiles for specifics blocks of time. Also, the core queue design can be applied in a wide variety of circumstances and is independent of the particular technologies used.

A careful reading of this document shows that the research goals were met successfully and that the queue design was effective. To begin with the design was made to be technology independent such that it could be implemented for any application on any architecture. It did indeed produce a system that handled several different types of tasks. Its performance was measured carefully and was found to be very satisfactory and vastly superior to previous non-queue methods. This improvement in performance translates directly into a hosting cost savings, which was one of the original stated goals of the research.

Nevertheless, it is restated here that the main goal was not performance in the sense that the queue was designed to produce maximum throughput. Rather, it was to preserve resources and to maximize throughput under resource constraints.

The design has great potential to assist many projects in preventing the need for developing separate queue systems for logically different task types. Instead, this research has demonstrated that a single generic queue core extended for any purpose can suffice. This saves significant development time, effort and costs. Because of the

ubiquitous need for queues, the usefulness of this research is far-reaching and it is hoped that it makes a solid contribution to the field of web application architecture.

Chapter 7 - Areas for Further Research and Development

During the time span allotted for this research, several potential areas for further research and development of the design and prototype were identified. These include the following items:

- Allowing for multiple matching profiles for a time span and choosing a single one based on criteria in order to increase the flexibility and preciseness of profiles.
- Expanding the scope of the design such that the queue itself becomes a very basic scheduler. This further decouples the application from its environment and allows for greater portability. For example, instead of having only a deadline for each task, the queue can also have a start time for each task. With both a start time and end time, the queue can aim to carry out a task within a time frame, thus achieving basic scheduling functionality. Furthermore, this idea can be adjusted to make tasks executed regularly rather than just once.
- Choosing profiles dynamically based on multiple factors in addition to time of day and day of week. The profile picking algorithm used in this research was limited to using time as a factor. However, other field types were also included in the profile table to allow for further development. For example, a profile could be picked by a combination of time of day as well as memory currently available. Whereas a default profile may, for example, direct the system to consume minimum resources during business hours, the system can be more intelligent to allow for exceptions. For example, a current lull in system activities by users may allow for higher resource consumption by the queue. This enhancement could result in a highly versatile queue with very controllable behavior.

One variation of this enhancement could be that the system uses real-time usage data to determine the desired queue throughput. For example, if there are many concurrent users, the system should pick a less demanding profile entry for messaging to preserve bandwidth. The ultimate test of success for this project is that the program should be run and two sets of data should be produced. One shows user traffic and the other shows email sending traffic. The latter should peak whenever the former drops.

- Logging system events in order to maintain records about queue activity and task errors. This information could also be periodically mailed to a system administrator for review.
- Assigning a due date to both tasks and jobs. The queue processing algorithm could then incorporate these values to determine which tasks to process first.
- Checking real-time resource usage and comparing it to multiple matching profiles and using a complex algorithm to pick from the matches. The algorithm could intelligently check which profile can most utilize currently available resources based on recorded usage trends. A further evolution beyond that could be using older historical trends to pick a profile in advance. However, in both scenarios, flexibility would have to be built in to allow the queue to cut back in resource usage if availability is scarce.
- Performing rigorous queue clean-up operations. As the queue processes tasks, the resources associated with those tasks should ideally be freed up immediately. These include temporary files, database records, etc. Although this is currently done to a certain extent, it is not optimized to be immediate and complete.

- Using materialized views for faster queue processing.
- Using high performance queue processing algorithms.

Beyond what has been mentioned, other areas specific to the application developed in this university scenario have also been identified as follows:

- Adding on other queue clients such as large score imports requiring many steps of validation.
- An enhancement could be allowing multiple recipients to be specified for each message in the queue. Currently, if multiple recipients are specified for a single message, a task is added for each recipient. However, this greatly affects the algorithms specified above since in this case there is no direct correlation between the number of tasks and the number of emails needed to be sent. It also introduces the complication that even a single message might exceed the number of emails per hour allowed constraint since the number of recipients might exceed this number.

Chapter 8 - References

Creswell, J. W. (2003). *Research design: qualitative, quantitative, and mixed methods approaches (2nd ed.)*. Thousand Oaks, CA: Sage Publications.

Leedy, P. D., & Ormrod, J. E. (2005). *Practical research: planning and design (8th ed.)*. Upper Saddle River, NJ: Pearson Education.

Willis, J. W. (2007). F*oundations of qualitative research: interpretive and critical approaches*. Thousand Oaks, CA: Sage Publications.

Yin, R. K. (2003). C*ase study research: design and methods (3rd ed.)*. Thousand Oaks, CA: Sage Publications.

Pasquali, M., Baraglia, R., Capannini, G., Ricci L., Laforenza, D. (2008). A two-level scheduler to dynamically schedule a stream of batch jobs in large-scale grids. *High performance distributed computing, proceedings of the 17th international symposium on high performance distributed computing*, p. 231-232.

Capannini, G., Baraglia, R., Puppin, D., Ricci, L., Pasquali, M. (2007). A job scheduling framework for large computing farms. *Proceedings of the 2007 ACM/IEEE conference on supercomputing*, Article 54.

Herrington, J. (2006). *Batch processing in PHP: how to create long-running jobs*. Retrieved from http://www.ibm.com/developerworks/opensource/library/os-php-batch on August 28, 2009.

Ronngren, R., Ayani, R. (1997). A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*. 7(2), 157 – 209.

March, S. and Smith, G. (1995). Design and natural science research on information technology. *Decision Support Systems*, 15, 251-266.

Hevner, A., March, S., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1), 75-105.

Vaishnavi, V. (2004). *Research patterns: improving and innovating information systems and technology*. Version 1.0, 2004. Georgia State University, Atlanta, GA.

Vaishnavi , V. and Kuechler, W. (2004). Design research in information systems. *ISWorld*. Retrieved on January 6, 2006 from http://www.isworld.org/Researchdesign/drisISworld.htm.

Kuechler, W. (2009). Email conversation.

Marc, S. and Smith, G. (1995). Design and natural science research on information technology. *Decision Support Systems*, 15, 251-266.

Configuring queue profiles. Retrieved from

http://www.juniper.net/techpubs/software/erx/junose82/swconfig-qos/html/queue-profiles-config2.html#1000063 on January 1, 2010.

DeVries, D., Naerezny, M. (2008). *Rails for PHP developers*. The Pragmatic Programmers.

Nicholas, J. (2004). *Project management for business and engineering*. Elsevier.

Oracle Corporation. *Oracle9i application developer's guide - advanced queuing. release 2 (9.2).* Part Number A96587-01 Introduction to Oracle Advanced Queuing.

Message-oriented middleware. Taken from http://en.wikipedia.org/wiki/Message-oriented_middleware on February 18, 2010.

Amotz Bar-Noy, Sudipto Guha, Yoav Katz, Joseph Seffi Naor, Baruch Schieber, And Hadas Shachnai. (2009). Throughput maximization of real-time scheduling with batching. *ACM Transactions on Algorithms*. 5(2), 1-17.

Berman, F., Wolski, R., Figueria, S., Schopf, J., Shao, G. (1996). Application-level scheduling on distributed heterogeneous networks. *Proceedings of Supercomputing*.

Blythe, J. Jain, S. Deelman, E. Gil, Y. Vahi, K. Mandal, A. Kennedy, K. Task scheduling strategies for workflow-based applications in grids. (2005). *Cluster Computing and the Grid*. 2, 759-767.

Zhang, H., Ferrari, D. Rate-controlled static-priority queuing. (1993). *Proceedings of IEEE INFOCOM'93*. 227 - 236.

Harchol-Balter, M, Bansal, N., Schroeder, B. (2000). Implementation of SRPT scheduling in web servers. *Technical Report CMU-CS-00-170*.

Cherkasova, L. (1998). Scheduling strategy to improve response time for web applications. *High-Performance Computing and Networking*. 1401, 305-314.

Sundell, H., Tsigas, P. (2005). Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*. 65(5), 609-627.

Abdelzaher, T., Bhatti, N. (1999). Web content adaptation to improve server overload behavior . *Computer Networks*. 31(11-16), 1563-1577.

Voigt, T., Tewari, R., Freimuth, D., Mehra, A. (2001). Kernel mechanisms for service differentiation in overloaded web servers. *Proceedings of the USENIX Annual Technical Conference*.

Schroeder, B., Harchol-Balter, M. (2006). Web servers under overload: how scheduling can help. *ACM Transactions on Internet Technology (TOIT)*. 6(1): 20 – 52.

Harchol-Balter, M., Schroeder. B., Bansal, N., Agrawal, M. (2003). Size-based scheduling to improve web performance . *ACM Transactions on Computer Systems (TOCS)*. 21(2): 207-233.

Schroeder, B., Harchol-Balter, M. (2006). Web servers under overload: how scheduling can help. *ACM Transactions on Internet Technology (TOIT)*. 6(1): 20-52.

Slothouber, L. (1996). A model of web server performance. *Proceedings of the Fifth International World Wide Web Conference*.

Schroeder, B., McWherter, D., Ailamaki, A. , Harchol-Balter, M. (2004). Priority mechanisms for OLTP and transactional web applications. International Conference on Data Engineering (ICDE'04). 20:535.

Advantages of using PL/PGSQL. PostgreSQL 8.4.2 Documentation Retrieved on March 11, 2010 from http://www.postgresql.org/docs/8.4/static/plpgsql-overview.html#PLPGSQL-ADVANTAGES.

Marchetti, P., Cerdá, J. (2009). An approximate mathematical framework for resource-constrained multistage. *Chemical Engineering Science*. 64(11): 2733-2748, 16p

Performance evaluation. (2007). *26th International Symposium on Computer, Performance, Modeling, Measurements, and Evaluation*. 64(9-12):1009-1028.

Abdelzaher, T., Bhatti, N. (1999). Adaptive content delivery for web server QoS. *International Workshop on Quality of Service*.

Enabling a web of billions of services. (2010). *Glossary*. Retrieved March 17, 2010 from http://www.soa4all.eu/glossary.html.

Appendix A - Class Interfaces

1. Generic Queue Core Classes

```
//Queue should be extended for new queue client types.
class Queue   {

      function Queue();

      //get pending jobs for a certain type
      function getPendingJobs(jt);

      function findDoneJobs();

      function closeDoneJobs();

      function getJobType(typNm);

      function delJobs();

      function delTasks();

      function initializeJob(typNm);

      function getNewJob(typNm);

      function getTaskType(typNm);

}

class JobType   {

      public job_type_ar;

      public job_type_id;

      function JobType(jobTypeId);

      //get list of users who should be notified that a job of this
      //type has been completed
      function getAlerts();

}

class Job   {

      //refresh fields in the active record object
      function refresh();

      function isLate();

      function isDone();

      function add();
```

```
        function setType(jt);

        // mark as complete
        function setAsDone();

        //decode data stored in info_json table field
        function getJSONInfo();

        function save();

        //is job done or not?
        function getStatus();

}

class Profile   {

        public profile_id;

        public profile_ar;

        function Profile(id);
}

class TaskType  {

        public task_type_id;

        public task_type_ar;

        public start_ts;

        public end_ts;

        function TaskType(id);

        //choose which performance profile to use
        function getActiveProfile();

}

//Task should be extended for new task types
abstract class Task  {

        public task_id;

        public task_ar;

        public start_time;

        public end_time;

        public status;
```

```
        public attempts;

        public curr_time;

        public task_type;

        public job_id;

        public now;

        private task_type_desc;

        function Task(id);

        //add to queue
        function enqueue();

        //mark as complete
        function setAsDone();

        //remove from queue
        function dequeue();

        //indicate that this task has been tried again
        function incrementAttempts();

        function getDefaultPriority();

        function setType(tt);

}
```

2. Enrollment Queue Classes

```
class EnrollmentQueue extends Queue {

        public qdEnrs;

        function EnrollmentQueue();

        function getEnrJobType();

        function getEnrTasksForJob(jb, onlyWithErrs);

        function getNumEnrsSent(start_ts, end_ts);

        function getPendingEnrs(num);

        //perform enrollment for all pending enrollments retrieved
        function processEnrs();

        function enqueEnr(sectionNumber, studentId, gender, campusId,
        offeringId, semesterId, jobId);

        function delEnrs();
```

```
}

class EnrFileQueue extends Queue {

	public qdEnrFiles;

	function EnrFileQueue();

	function getEnrFileJobType();

	function addFile(path, offId, campId, jobId);

	//get queued enrollments for a single job
	function getEnrFileTasksForJob(jb, onlyWithErrs);

	function getNumEnrFilesSent(start_ts, end_ts);

	function getPendingEnrFiles(num);

	function processEnrFiles();

	function enqueEnrFile(sectionNumber, studentId, gender, campusId,
	offeringId, semesterId, jobId);

	function delEnrFiles();

}

class QueuedEnrFile extends Task {

	public enrFile_id;

	public enr_file_ar;

	function QueuedEnrFile(id);

	function read();

}

class EnrJob  extends Job {

	public enr_ar;

	public job_id;

	function EnrJob(jobId);

}

class QueuedEnrollment extends Task {

	public qd_enrollment_id;
```

```
        public qd_enrollment_ar;
        function QueuedEnrollment(id);
        function enroll();
}
```

3. Message Queue Classes

```
class MessageQueue extends Queue {
 public msgs;
 function MessageQueue();
 function getUnsentMsgs(num);
 function getNumMsgsSent(start_ts, end_ts);
 function sendMsgs();
 function delMsgs();
}
class Message extends Task {
 private message_id;
 public msg_ar;
 public body_path;
 public body_tpl;
 public title;
 public from;
 public to;
 public to_user_id;
 function Message(id);
 function send();
 function addRecipients(recps);
 //add to queue
 function addMsg();
 function del();
```

```
 //return array of file paths for attachments
 function getAttchStr();

 function hasAttch();

}
```

Appendix B - SQL Table Definitions

1. Queue Tables

```
CREATE TABLE profiles
(
  profile_id integer NOT NULL,
  start_time time,
  end_time time ,
  sat boolean,
  sun boolean,
  mon boolean,
  tue boolean,
  wed boolean,
  thu boolean,
  fri boolean,
  task_type_id integer,
  max_per_hour integer,
  max_per_sess integer,
  default_priority integer,
  CONSTRAINT profile_pk PRIMARY KEY (profile_id)
)

CREATE TABLE task_types
(
  task_type_id integer NOT NULL,
  task_type_name character varying,
  default_priority integer,
  default_duration integer,
  max_attempts integer,
  CONSTRAINT task_types_pk PRIMARY KEY (task_type_id)
)

CREATE TABLE tasks
(
  task_id integer NOT NULL,
  task_type_id integer,
  job_id integer,
  start_ts timestamp ,
  end_ts timestamp ,
  priority smallint,
  attempts integer,
  status character(1),
  user_id integer,
  message text,
  success character(1),
  CONSTRAINT tasks_pk PRIMARY KEY (task_id),
  CONSTRAINT tasks_job_id_fk FOREIGN KEY (job_id)
      REFERENCES jobs (job_id)
      ON UPDATE CASCADE ON DELETE RESTRICT,
```

```
  CONSTRAINT tasks_type_fk FOREIGN KEY (task_type_id)
      REFERENCES task_types (task_type_id)
      ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE jobs
(
  job_id integer NOT NULL,
  job_type_id integer,
  status character(1),
  user_id integer,
  priority integer,
  start_ts timestamp ,
  end_ts timestamp ,
  due_date_ts timestamp ,
  title text,
  description text,
  info_json text,
  CONSTRAINT jobs_pk PRIMARY KEY (job_id),
  CONSTRAINT jobs_type_fk FOREIGN KEY (job_type_id)
      REFERENCES job_types (job_type_id)
      ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE job_types
(
  job_type_id integer NOT NULL,
  job_name character varying,
  default_task_duration integer,
  default_task_priority integer,
  max_task_attempts integer,
  max_tasks_per_hour integer,
  max_tasks_per_sess integer,
  CONSTRAINT job_types_pk PRIMARY KEY (job_type_id)
)

CREATE TABLE job_alerts
(
  job_alert_id integer NOT NULL,
  employee_id integer,
  job_type_id integer,
  CONSTRAINT job_alerts_pk PRIMARY KEY (job_alert_id)
)
```

2. Queue Client Tables

```
CREATE TABLE messages
(
  message_id integer NOT NULL,
  body text,
```

```
  from_user_id integer,
  title text,
  to_user_id integer,
  from_addr character varying(200),
  to_addr character varying(200),
  body_path character varying(2000),
  was_sent boolean DEFAULT false,
  attachments character varying,
  task_id integer,
  CONSTRAINT msg_pk PRIMARY KEY (message_id),
  CONSTRAINT msg_tsk_id_fk FOREIGN KEY (task_id)
      REFERENCES tasks (task_id)
      ON UPDATE CASCADE ON DELETE CASCADE
)

CREATE TABLE queued_enrollments
(
  qd_enrollment_id integer NOT NULL,
  sec_num integer,
  gender character(1),
  student_id bigint,
  campus_id integer,
  task_id bigint,
  oid integer,
  offering_id bigint,
  sec_num_source character(1),
  max_sec_size integer,
  CONSTRAINT qd_enrs_pk PRIMARY KEY (qd_enrollment_id),
  CONSTRAINT qd_enrs_task_id_fk FOREIGN KEY (task_id)
      REFERENCES tasks (task_id)
      ON UPDATE CASCADE ON DELETE CASCADE
)

CREATE TABLE enrollment_files
(
  enr_file_id integer NOT NULL,
  path character varying,
  offering_id integer,
  campus_id integer,
  job_id integer,
  curr_line integer DEFAULT 0,
  CONSTRAINT enr_file_pk PRIMARY KEY (enr_file_id),
  CONSTRAINT enr_file_job_id_fk FOREIGN KEY (job_id)
      REFERENCES jobs (job_id)
      ON UPDATE CASCADE ON DELETE CASCADE
)
```

Glossary

Generic Queue Core Terms

task – an entry in the queue referring to a unit of work that must be processed and completed.

task type – the type designation of a task. This determines the processing logic for a task.

job – a logical grouping of tasks usually created during the same execution cycle or close together in time. For example, a weekly student report is generated for 10,000 students. These are all submitted to the queue at once as members of the same job. If all 10,000 reports have been mailed out, that single job has been completed.

task priority – a numerical value indicating the relative priority of a task.

job priority – a numerical value indicating the relative priority of a job.

queue client – Also known as a *Job Processor***,** an application component that makes use of the queue services and contains business logic that processes all of the tasks of a given job. Examples are email messaging and enrollment clients.

profile entry – a directive stored in the database indicating how the queue should behave under a set of environmental circumstances for a given task type. Each entry contains ideal queue throughput values for a given time range and set of days of the week. An example is that during the weekend at any time the queue should send 100 emails per hour. Another profile entry could indicate that during the weekdays from 9 AM to 5 PM the queue should send only 5 per hour.

profile – a set of profile entries for a task type. An example is a profile for messaging that contains two entries. One is for work hours, another is for off hours during weekdays and a third is for the weekend.

active profile – the profile that has been chosen to be used for a given moment in time. The other profiles are *inactive*.

processing cycle – a single execution run of a queue client (see above). Each processing cycle is invoked by the system scheduler.

system scheduler – the mechanism that determines how often and when to run the queue client processing cycles for each job type.

Application Specific Terms

These are terms relating to the specific queue clients developed for the instantiated prototype.

enrollment – the placement of a student into a course section.

enrollment file – a file containing one or more lines with each line containing the data needed to enroll a single student into a section